Sercos Objects

Introduction

A **Sercos** object manages an individual SERCOS master.

The **SERCOS** (**S**erial **R**eal-**T**ime **C**ommunication **S**ystem) protocol was designed specifically for the motion control industry. The protocol uses a ring topology which allows communication to occur at specific assigned times within a predetermined cycle. Thus, SERCOS is deterministic allowing the Master (control unit) to synchronize command and feedback values to and from all of the Slaves (drives).

The MPI supports the use of SERCOS fiberoptic technology. The XMP-SERCOS-CPCI supports up to 3 SERCOS rings with support for up to 24 drives, in a 2-slot configuration in a CompactPCI form-factor. The MPI includes 4 SERCOS object types: SERCOS, NODE, IDN, and IDNLIST. Utility programs for initializing a ring and reading or writing IDNs are also provided with the software distribution.

Only a limited number of SERCOS drive manufacturers are supported in the MPI. For a list of supported drives, click here. Contact MEI for more information about supported SERCOS drives.

Introduction | Overview | Data Types | Communications
Procedures | Telegrams | Topologies

Methods

Create, Delete, Validate Methods

mpiSercos**Create**	Create Sercos object
mpiSercos**Delete**	Delete Sercos object
mpiSercos**Validate**	Validate Sercos object

Configuration and Information Methods

mpiSercos**ConfigGet**	Get Sercos object's configuration
mpiSercos**ConfigSet**	Set Sercos object's configuration
mpiSercos**Error**	Set contents of error with platform-specific info (about the error)
mpiSercos**FlashConfigGet**	Get flash config of Sercos
mpiSercos**FlashConfigSet**	Set flash config of Sercos
meiSercos**ServiceIdnFieldGet**	
meiSercos**ServiceIdnFieldSet**	
meiSercos**ServiceProcedure**	
mpiSercos**Status** / meiSercos**Status**	Get the status of a Sercos object

Action Methods

mpiSercos**Init**	Initialize a Sercos object to a specified SERCOS phase
mpiSercos**Reset**	Reset a Sercos object to SERCOS Phase 0

Memory Methods

mpiSercos**Memory**	Set Sercos memory address
mpiSercos**MemoryGet**	Copy bytes of Sercos memory to application memory
mpiSercos**MemorySet**	Copy bytes of application memory to Sercos memory

Relational Methods

mpiSercos**Control**	Get handle to Control object that a Sercos object is associated with
mpiSercos**Number**	Get the index of a Sercos object from the Control object that the Sercos object is associated with

List Methods for Sercos Nodes

mpiSercos**Node**	Get handle to the indexth Node in a Sercos object's list
mpiSercos**NodeAppend**	Append a Node object to a Sercos object's list
mpiSercos**NodeCount**	Get the number of Node objects in a Sercos object's list
mpiSercos**NodeFirst**	Get handle to the first Node in a Sercos object's list
mpiSercos**NodeIndex**	Get the index of a Node object in a Sercos object's list
mpiSercos**NodeInsert**	Insert a Node into a Sercos object's list
mpiSercos**NodeLast**	Get handle to the last Node in a Sercos object's list
mpiSercos**NodeListGet**	Get the list of Nodes associated with a Sercos object
mpiSercos**NodeListSet**	Create a list of Nodes associated with a Sercos object
mpiSercos**NodeNext**	Get handle to the next Node in a Sercos object's list
mpiSercos**NodePrevious**	Get handle to the previous Node in a Sercos object's list
mpiSercos**NodeRemove**	Remove a Node from a Sercos object's list

Data Types

MPISercos**Baud**
MPISercos**Error**
MPISercos**ErrorGroup**
MPISercos**ErrorType**
MPISercos**LoopStatus**
MPISercos**Message** / MEISercos**Message**
MPISercos**ProcedureAction**
MEISercos**ServiceContainer**
MPISercos**Status**

Constants

MPISercos**NODE_COUNT_MAX**

Macros

mpiSercos**NodeIdnDataSET**
mpiSercos**NodeIdnGET**

mpiSercosCreate

Declaration

```
const MPISercos mpiSercosCreate(MPIControl control,
                                long       number)
```

Required Header

stdmpi.h

Description

SercosCreate creates a Sercos object that is associated with the SERCON 410B chip, which in turn is identified by ***number*** on a motion controller (***control***). *SercosCreate* is the equivalent of a C++ constructor.

Return Values

handle	to a Sercos object (associated with ***number*** and ***control***)
MPIHandleVOID	if the object could not be created

See Also

mpiSercosDelete | mpiSercosValidate

mpiSercosDelete

Declaration `long mpiSercosDelete(MPISercos sercos)`

Required Header stdmpi.h

Description **SercosDelete** deletes a Sercos object and invalidates its handle (***sercos***). *SercosDelete* is the equivalent of a C++ destructor.

Return Values	
MPIMessageOK	if *SercosDelete* successfully deletes the Sercos object and invalidates its handle

See Also mpiSercosCreate | mpiSercosValidate

mpiSercosValidate

Declaration `long mpiSercosValidate(MPISercos sercos)`

Required Header stdmpi.h

Description **SercosValidate** validates the Sercos object and its handle (***sercos***).

Return Values	
MPIMessageOK	if Sercos is a handle to a valid object.

See Also mpiSercosCreate | mpiSercosDelete

mpiSercosConfigGet

Declaration

```
long mpiSercosConfigGet(MPISercos       sercos,
                        MPISercosConfig *config,
                        void            *external)
```

Required Header

stdmpi.h

Description

SercosConfigGet gets a Sercos object's (***sercos***) configuration and writes it into the structure pointed to by ***config***, and also writes it into the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is *in addition* to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only ***external*** either points to a structure of type MEISercosConfig{} or is NULL.

Return Values

MPIMessageOK	if *SercosConfigGet* successfully writes the Sercos object's configuration to the structure(s)

See Also

mpiSercosConfigSet

mpiSercosConfigSet

Declaration

```
long mpiSercosConfigSet(MPISercos       sercos,
                        MPISercosConfig *config,
                        void            *external)
```

Required Header

stdmpi.h

Description

SercosConfigSet sets a Sercos object's (***sercos***) configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The configuration information in ***external*** is in addition to the configuration information in ***config***, i.e, the configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only — ***external*** either points to a structure of type MEISercosConfig{} or is NULL.

Return Values

MPIMessageOK	if *SercosConfigSet* successfully sets the Sercos object's configuration using data from the structure(s)

See Also

mpiSercosConfigGet

mpiSercosError

Declaration

```
long mpiSercosError(MPISercos      sercos,
                    MPISercosError *error)
```

Required Header

stdmpi.h

Description

SercosError reads the error code from a Sercos object and writes it into the location pointed to by ***error***.

sercos	a handle to a Sercos object
***error**	a pointer to a MPISercosError

Return Values

MPIMessageOK	if *mpiSercosError* successfully writes the current ***sercos*** error to ***error***.

See Also

mpiSercosFlashConfigGet

Declaration

```
long mpiSercosFlashConfigGet(MPISercos        sercos,
                             void             *flash,
                             MPISercosConfig  *config,
                             void             *external)
```

Required Header

stdmpi.h

Description

SercosFlashConfigGet gets a Sercos object's (***sercos***) flash configuration and writes it in the structure pointed to by ***config***, and also writes it in the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sercos object's flash configuration information in ***external*** is in addition to the Sercos object's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that config or ***external*** can be NULL (but not both NULL).

XMP Only	***external*** either points to a structure of type **MEISercosConfig{}** or is NULL.

Return Values

MPIMessageOK	if *SercosFlashConfigGet* successfully writes the Sercos object's flash configuration to the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiSercosFlashConfigSet

mpiSercosFlashConfigSet

Declaration

```
long mpiSercosFlashConfigSet(MPISercos        sercos,
                             void             *flash,
                             MPISercosConfig  *config,
                             void             *external)
```

Required Header

stdmpi.h

Description

SercosFlashConfigSet sets a Sercos object's (***sercos***) flash configuration using data from the structure pointed to by ***config***, and also using data from the implementation-specific structure pointed to by ***external*** (if ***external*** is not NULL).

The Sercos object's flash configuration information in ***external*** is in addition to the Sercos object's flash configuration information in ***config***, i.e, the flash configuration information in ***config*** and in ***external*** is not the same information. Note that ***config*** or ***external*** can be NULL (but not both NULL).

XMP Only

external either points to a structure of type **MEISercosConfig{}** or is NULL.

Return Values

MPIMessageOK	if *SercosFlashConfigSet* successfully sets the Sercos object's flash configuration using data from the structure(s) ***flash*** is either an MEIFlash handle or MPIHandleVOID. If ***flash*** is MPIHandleVOID, an MEIFlash object will be created and deleted internally.

See Also

MEIFlash | mpiSercosFlashConfigGet

meiSercosServiceIdnFieldGet

Declaration

```
long meiSercosServiceIdnFieldGet(MPISercos                  sercos,
                                 MEISercosServiceContainer *service,
                                 MPIIdn                     idn,
                                 MPIIdnField                field)
```

Required Header

stdmei.h

Description

SercosServiceIdnFieldGet uses a service container to read the specified ***field*** from an ***idn***, and write it into the idn object.

sercos	a handle to a Sercos object
***service**	a pointer to a service container
idn	a handle to an Idn object
field	enumeration corresponding to each element in the structure MPIIdnElement

Return Values

MPIMessageOK	if *SercosServiceIdnFieldGet* successfully reads the field from the service container.

See Also

mpiSercosServiceIdnFieldSet | MPIIdnElement

meiSercosServiceIdnFieldSet

Declaration

```
long meiSercosServiceIdnFieldSet(MPISercos                  sercos,
                                 MEISercosServiceContainer *service,
                                 MPIIdn                     idn,
                                 MPIIdnField                field)
```

Required Header

stdmei.h

Description

SercosServiceIdnFieldSet uses a service container to write the specified ***field*** from an idn object to an ***idn***.

sercos	a handle to a Sercos object
***service**	a pointer to a service container
idn	a handle to an Idn object
field	enumeration corresponding to each element in the structure MPIIdnElement

Return Values

MPIMessageOK	if *SercosServiceIdnFieldSet* successfully writes the field to an idn via the service container.

See Also

mpiSercosServiceIdnFieldGet | MPIIdnElement

meiSercosServiceProcedure

Declaration

```
long meiSercosServiceProcedure(MPISercos                  sercos,
                               MEISercosServiceContainer *service,
                               MPISercosProcedureAction   action,
                               MPIIdn                     idn,
                               unsigned long             *status)
```

Required Header

stdmei.h

Description

SercosServiceProcedure uses a service container to send a procedure ***action*** to a sercos ***idn***.

sercos	a handle to a Sercos object
***service**	a pointer to a service container
action	a command used to invoke a procedure
idn	a handle to an Idn object
***status**	a pointer to an unsigned long

Return Values

MPIMessageOK	if *SercosServiceProcedure* successfully sends a procedure action via a service container.

See Also

MPISercosProcedureAction

mpiSercosStatus / meiSercosStatus

mpiSercosStatus

Declaration

```
long mpiSercosStatus(MPISercos        sercos,
                     MPISercosStatus  *status,
                     void             *external)
```

Required Header stdmpi.h

Description **SercosStatus** writes the status of a Sercos object (***sercos***) to the structure pointed to by ***status***.

Return Values	
MPIMessageOK	if *SercosStatus* successfully writes the status of a Sercos object to the structure

meiSercosStatus

Declaration

```
long meiSercosStatus(MPIControl        control,
                     MEIMotorTypeInfo  *motorInfo,
                     MPISercosStatus   *status)
```

Required Header stdmei.h

Description **SercosStatus** reads the status from a ***control*** and writes it into the location pointed to by ***status***. The status is read from the nodeNumber specified in the structure pointed to by ***motorInfo***.

control	a handle to the Sercos object.
***motorInfo**	a pointer to a MEIMotorTypeInfo structure
***status**	a handle to a Node object.

Return Values	
MPIMessageOK	if *SercosStatus* successfully reads the status.

See Also

mpiSercosInit

Declaration

```
long mpiSercosInit(MPISercos sercos,
                   long      phase)
```

Required Header

stdmpi.h

Description

SercosInit advances the initialization of a Sercos object (***sercos***) to a specific phase (***phase***).

If ***phase*** = -1, then the Sercos object is moved to Phase 4 (full initialization). If the current phase number (of the Sercos object) is greater than ***phase***, initialization restarts at Phase 0 and advances to Phase ***phase***.

Return Values

MPIMessageOK	if *SercosInit* successfully initializes the state of a Sercos object (***sercos***) to the Phase ***phase***

See Also

mpiSercosReset

Declaration `long mpiSercosReset(MPISercos sercos)`

Required Header stdmpi.h

Description **SercosReset** resets a Sercos object (***sercos***), thereby setting its initialization phase to 0.

Return Values

MPIMessageOK	if *SercosReset* successfully resets the Sercos object

See Also

mpiSercosMemory

Declaration

```
long mpiSercosMemory(MPISercos sercos,
                     void      **memory)
```

Required Header

stdmpi.h

Description

SercosMemory writes an address [that is used to access Sercos (sercos) memory] to the contents of ***memory***. This address (or an address calculated from it) is passed as the ***src*** argument to mpiSercosMemoryGet(...) and as the ***dst*** argument to mpiSercosMemorySet(...).

Return Values

MPIMessageOK	if *SercosMemory* successfully writes the Sercos memory address to the contents of ***memory***

See Also

mpiSercosMemoryGet | mpiSercosMemorySet

mpiSercosMemoryGet

Declaration

```
long mpiSercosMemoryGet(MPISercos sercos,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

SercosMemoryGet copies ***count*** bytes of Sercos (***sercos***) memory (starting at address ***src***) to application memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *SercosMemoryGet* successfully copies count bytes of Sercos memory to application memory

See Also

mpiSercosMemorySet | mpiSercosMemory

mpiSercosMemorySet

Declaration

```
long mpiSercosMemorySet(MPISercos sercos,
                        void      *dst,
                        void      *src,
                        long      count)
```

Required Header

stdmpi.h

Description

SercosMemorySet copies ***count*** bytes of application memory (starting at address ***src***) to Sercos (***sercos***) memory (starting at address ***dst***).

Return Values

MPIMessageOK	if *SercosMemorySet* successfully copies count bytes of application memory to Sercos memory

See Also

mpiSercosMemoryGet | mpiSercosMemory

mpiSercosControl

Declaration `const MPIControl mpiSercosControl(MPISercos sercos)`

Required Header stdmpi.h

Description **SercosControl** returns a handle to the Control object with which the Sercos object is associated.

sercos	a handle to the Sercos object

Return Values

handle	to the Control object that a Sercos object (***sercos***) is associated with
MPIHandleVOID	if ***sercos*** is invalid

See Also mpiSercosCreate | mpiControlCreate

mpiSercosNumber

Declaration

```
long mpiSercosNumber(MPISercos sercos,
                     long      *number)
```

Required Header

stdmpi.h

Description

SercosNumber writes the index of a Sercos object (***sercos***, on the motion controller that the Sercos object is associated with) to the location pointed to by ***number***.

Return Values

MPIMessageOK	if *SercosNumber* successfully writes the Sercos object's index to the location

See Also

mpiSercosNode

Declaration

```
const MPINode  mpiSercosNode(MPISercos sercos,
                             long      index)
```

Required Header

stdmpi.h

Description

SercosNode returns the element at the position on the list indicated by ***index***.

sercos	a handle to the Sercos object.
index	a position in the list.

Return Values

handle	to the ***index***th Node object of a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***index*** is less than 0 if ***index*** is greater than **mpiSercosCount**(***sercos***) if ***index*** is equal to **mpiSercosCount**(***sercos***)
MPIMessageARG_INVALID	if ***index*** is a negative number.
MEIListMessageELEMENT_NOT_FOUND	if ***index*** is greater than or equal to the number of elements in the list.
MPIMessageHANDLE_INVALID	if ***sercos*** is an invalid handle.

See Also

mpiSercosNodeAppend

Declaration

```
long mpiSercosNodeAppend(MPISercos sercos,
                         MPINode   node)
```

Required Header

stdmpi.h

Description

SercosNodeAppend appends a Node object (***node***) to a Sercos object (***sercos***).

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

MPIMessageOK	if *SercosNodeAppend* successfully appends the Node object to the Sercos object
MPIMessageHANDLE_INVALID	Either ***sercos*** or ***node*** is an invalid handle.
MPIMessageNO_MEMORY	Not enough memory was available.

See Also

mpiSercosNodeCount

Declaration

```
long mpiSercosNodeCount(MPISercos sercos)
```

Required Header

stdmpi.h

Description

SercosNodeCount returns the number of elements on the list.

sercos	a handle to the Sercos object.

Return Values

number	of Node objects in a Sercos object (***sercos***)
-1	if ***sercos*** is invalid
0	if ***sercos*** is empty

See Also

mpiSercosNodeFirst

Declaration

```
const MPINode mpiSercosNodeFirst(MPISercos sercos)
```

Required Header

stdmpi.h

Description

SercosNodeFirst returns the first element in the list. This function can be used in conjuntion with mpiSercosNodeNext() in order to iterate through the list.

sercos	a handle to the Sercos object.

Return Values

handle	to the first Node object of a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***sercos*** is empty
MPIMessageHANDLE_INVALID	if ***sercos*** is an invalid handle.

See Also

mpiSercosNodeNext | mpiSercosNodeLast

mpiSercosNodeIndex

Declaration

```
long mpiSercosNodeIndex(MPISercos sercos,
                        MPINode node)
```

Required Header

stdmpi.h

Description

SercosNodeIndex returns the position of ***node*** on the list.

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

index	of a Node object (***node***) in a Sercos object (***sercos***)
-1	if ***sercos*** is invalid if ***node*** was not found in ***sercos***

See Also

mpiSercosNodeInsert

Declaration

```
long mpiSercosNodeInsert(MPISercos sercos,
                         MPINode   node,
                         MPINode   insert)
```

Required Header

stdmpi.h

Description

SercosNodeInsert inserts a Node object (***insert***) in a Sercos object (***sercos***), just after the specified Node object (***node***).

Return Values

MPIMessageOK	if *SercosNodeInsert* successfully inserts the Node object (***insert***) just after the specified Node object (***node***), in the Sercos object (***sercos***)

See Also

mpiSercosNodeLast

Declaration

```
const MPINode mpiSercosNodeLast(MPISercos sercos)
```

Required Header

stdmpi.h

Description

SercosNodeLast returns the last element in the list. This function can be used in conjuntion with mpiSercosNodePrevious() in order to iterate through the list backwards.

sercos	a handle to the Sercos object.

Return Values

handle	to the last Node object (***node***) of a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***sercos*** is empty
MPIMessageHANDLE_INVALID	if ***sercos*** is an invalid handle.

See Also

mpiSercosNodePrevious | mpiSercosNodeFirst

mpiSercosNodeListGet

Declaration

```
long mpiSercosNodeListGet(MPISercos sercos,
                          long      *nodeCount,
                          MPINode   *nodeList)
```

Required Header

stdmpi.h

Description

SercosNodeListGet gets the Nodes in a Sercos object (***sercos***), and writes an array (of handles to those Nodes, size= ***nodeCount***) to the location pointed to by ***nodeList***, and also writes the number of Nodes (in the Sercos object) to the location pointed to by ***nodeCount***.

Return Values

MPIMessageOK	if *SercosNodeListGet* successfully writes the array of Node handles and the number of Nodes to the 2 locations (***nodeList, nodeCount***)

See Also

mpiSercosNodeListSet

mpiSercosNodeListSet

Declaration

```
long mpiSercosNodeListSet(MPISercos sercos,
                          long      nodeCount,
                          MPINode   *nodeList)
```

Required Header

stdmpi.h

Description

Using the Node handles specified by ***nodeList***, **SercosNodeListSet** creates a list of Nodes (length=***nodeCount***). Any existing Node list of the Sercos object (***sercos***) is completely replaced.

The ***nodeList*** argument is the address of an array of ***nodeCount*** Node handles, or is NULL (if ***nodeCount*** is equal to zero).

You can also create a Node list incrementally (one Node at a time), by using the **mpiNodeListAppend(...)** and/or **mpiNodeListInsert(...)** methods. You can use any *mpiNodeList* method to examine and manipulate a Node list sequence, regardless of how the the Node list was created.

Return Values

MPIMessageOK	if *IdnListSet* successfully creates the IdnList using the handles in ***idnArray***

See Also

mpiSercosNodeListGet

mpiSercosNodeNext

Declaration

```
const MPINode  mpiSercosNodeNext(MPISercos sercos,
                                 MPINode   node)
```

Required Header

stdmpi.h

Description

SercosNodeNext returns the next element following "node" on the list. This function can be used in conjuntion with mpiSercosNodeFirst() in order to iterate through the list.

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

handle	to the Node object that is just after the specified Node object (***node***), in a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***node*** is the last Node in the Sercos object (***sercos***)
MPIMessageHANDLE_INVALID	Either ***sercos*** or ***node*** is an invalid handle.

See Also

mpiSercosNodeFirst | mpiSercosNodePrevious

mpiSercosNodePrevious

Declaration

```
const MPINode  mpiSercosNodePrevious(MPISercos sercos,
                                     MPINode   node)
```

Required Header

stdmpi.h

Description

SercosNodePrevious returns the previous element prior to "node" on the list. This function can be used in conjuntion with mpiSercosNodeLast() in order to iterate through the list backwards.

sercos	a handle to the Sercos object.
node	a handle to a Node object.

Return Values

handle	to the Node object that is just before the specified Node object (node), in a Sercos object (***sercos***)
MPIHandleVOID	if ***sercos*** is invalid if ***node*** is the first Node in the Sercos object (***sercos***)
MPIMessageHANDLE_INVALID	Either ***sercos*** or ***node*** is an invalid handle.

See Also

mpiSercosNodeLast | mpiSercosNodeNext

mpiSercosNodeRemove

Declaration

```
long mpiSercosNodeRemove(MPISercos sercos,
                         MPINode   node)
```

Required Header stdmpi.h

Description **SercosNodeRemove** removes a Node object (***node***) from a Sercos object (***sercos***).

Return Values	
MPIMessageOK	if *SercosNodeRemove* successfully removes the Node object from the Sercos object

See Also

MPISercosBaud

MPISercosBaud

```
typedef enum {
    MPISercosBaudINVALID,

    MPISercosBaud2MBIT,
    MPISercosBaud4MBIT,
    MPISercosBaud10MBIT,

} MPISercosBaud;
```

Description

SercosBaud defines the different baud rates that the SERCOS ring can communicate at.

See Also

MPISercosError

MPISercosError

```
typedef enum {
    MPISercosErrorSHIFT,

    MPISercosErrorGENERAL,
    MPISercosErrorNONE,
    MPISercosErrorCHANNEL_NOT_OPEN,
    MPISercosErrorCHANNEL_ACCESS,

    MPISercosErrorIDN,
    MPISercosErrorNO_IDN,
    MPISercosErrorIDN_ACCESS,

    MPISercosErrorNAME,
    MPISercosErrorNO_NAME,
    MPISercosErrorNAME_TOO_SHORT,
    MPISercosErrorNAME_TOO_LONG,
    MPISercosErrorNAME_NO_CHANGE,
    MPISercosErrorNAME_WRITE_PROTECT,

    MPISercosErrorATTR,
    MPISercosErrorATTR_TOO_SHORT,
    MPISercosErrorATTR_TOO_LONG,
    MPISercosErrorATTR_NO_CHANGE,
    MPISercosErrorATTR_WRITE_PROTECT,

    MPISercosErrorUNIT,
    MPISercosErrorNO_UNIT,
    MPISercosErrorUNIT_TOO_SHORT,
    MPISercosErrorUNIT_TOO_LONG,
    MPISercosErrorUNIT_NO_CHANGE,
    MPISercosErrorUNIT_WRITE_PROTECT,

    MPISercosErrorMIN,
    MPISercosErrorNO_MIN,
    MPISercosErrorMIN_TOO_SHORT,
    MPISercosErrorMIN_TOO_LONG,
    MPISercosErrorMIN_NO_CHANGE,
    MPISercosErrorMIN_WRITE_PROTECT,

    MPISercosErrorMAX,
    MPISercosErrorNO_MAX,
    MPISercosErrorMAX_TOO_SHORT,
    MPISercosErrorMAX_TOO_LONG,
    MPISercosErrorMAX_NO_CHANGE,
    MPISercosErrorMAX_WRITE_PROTECT,

    MPISercosErrorOP,
    MPISercosErrorOP_TOO_SHORT,
```

```
        MPISercosErrorOP_TOO_LONG,
        MPISercosErrorOP_NO_CHANGE,
        MPISercosErrorOP_WRITE_PROTECT,
        MPISercosErrorOP_MIN,
        MPISercosErrorOP_MAX,
        MPISercosErrorOP_DATA,
        MPISercosErrorOP_PASSWORD,
    } MPISercosError;
```

Description

The **SercosError** enumeration defines all the different errors that can occur in SERCOS.

See Also

For a more in-depth breakdown of the errors, please refer to a SERCOS Specification Manual.

MPISercosErrorGroup

MPISercosErrorGroup

```
typedef enum {
    MPISercosErrorGroupINVALID,

    MPISercosErrorGroupGENERAL,
    MPISercosErrorGroupIDN,
    MPISercosErrorGroupNAME,
    MPISercosErrorGroupATTR,
    MPISercosErrorGroupUNIT,
    MPISercosErrorGroupMIN,
    MPISercosErrorGroupMAX,
    MPISercosErrorGroupOP,

} MPISercosErrorGroup;
```

Description

When an error occurs, **SercosErrorGroup** defines what part of the IDN field has the error.

See Also

MPISercosErrorType

MPISercosErrorType

```
typedef enum {
    MPISercosErrorTypeNONE,
    MPISercosErrorTypeDOES_NOT_EXIST,
    MPISercosErrorTypeTOO_SHORT,
    MPISercosErrorTypeTOO_LONG,
    MPISercosErrorTypeNO_CHANGE,
    MPISercosErrorTypeWRITE_PROTECT,
    MPISercosErrorTypeTOO_SMALL,
    MPISercosErrorTypeTOO_BIG,
    MPISercosErrorTypeINVALID_DATA,
    MPISercosErrorTypeINVALID_ACCESS,
    MPISercosErrorTypePASSWORD_PROTECT,
} MPISercosErrorType;
```

Description

SercosErrorType enumeration contains basic SERCOS error types that are ORed into the MPISercosErrorGroup enumeration to construct unique MPISercosError codes.

MPISercosErrorTypeNONE	This error type indicates no error occurred.
MPISercosErrorTypeDOES_NOT_EXIST	This error type indicates that a service channel is not open, an idn does not exist or an idn element does not exist.
MPISercosErrorTypeTOO_SHORT	This error type indicates an idn element is too short.
MPISercosErrorTypeTOO_LONG	This error type indicates an idn element is too long.
MPISercosErrorTypeNO_CHANGE	This error type indicates an idn element cannot be changed.
MPISercosErrorTypeWRITE_PROTECT	This error type indicates an idn element is read only.
MPISercosErrorTypeTOO_SMALL	This error type indicates the idn operation data is too small.
MPISercosErrorTypeTOO_BIG	This error type indicates the idn operation data is too big.
MPISercosErrorTypeINVALID_DATA	This error type indicates the idn operation data is not valid.
MPISercosErrorTypeINVALID_ACCESS	This error type indicates a service channel is not accessible or an idn is not accessible.
MPISercosErrorTypePASSWORD_PROTECT	This error type indicates the idn operation data is password protected.

See Also

MPISercosError

MPISercosLoopStatus

MPISercosLoopStatus

```
typedef enum {
     MPISercosLoopStatusOPEN,
} MPISercosLoopStatus;
```

Description

SercosLoopStatus defines the status of a SERCOS loop.

See Also

MPISercosMessage / MEISercosMessage

MPISercosMessage

```
typedef enum {

    MPISercosMessageSERCOS_INVALID,
    MPISercosMessageHANDSHAKE_TIMEOUT,
    MPISercosMessageIDN_FIELD_INVALID,
    MPISercosMessageMST_RECEIVE_ERROR,
    MPISercosMessageNODE_NOT_FOUND,
    MPISercosMessagePROCEDURE_DATA_INVALID,
    MPISercosMessagePROCEDURE_ERROR,
    MPISercosMessagePROTOCOL_ERROR,
    MPISercosMessageRING_NOT_CLOSED,
    MPISercosMessageSAMPLE_RATE_INVALID,
    MPISercosMessageSERVICE_CHANNEL_BUSY,
    MPISercosMessageSERVICE_CHANNEL_ERROR,
    MPISercosMessagePROCEDURE_TIMEOUT,
    MPISercosMessageM_BUSY_TIMEOUT,
} MPISercosMessage;
```

Description

MPISercosMessageSERCOS_INVALID	
Meaning	The specified SERCOS ring is not valid.
Possible Causes	A number passed to MPISercosCreate( ) is larger than the maximum allowable number of Sercos objects.
Recommendations	If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageHANDSHAKE_TIMEOUT	
Meaning	A timeout has occurred during a SERCOS service container operation.
Possible Causes	If there is some sort of communication error, the handshaking will probably not be synchronized, and a timeout will occur.
Recommendations	If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageIDN_FIELD_INVALID	
Meaning	A user is trying to read an IDN field that does not exist.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageMST_RECEIVE_ERROR	
Meaning	.The wrong data type is being used to access an IDN field.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageNODE_NOT_FOUND	
Meaning	The specified Node number is invalid.
Possible Causes	In the phase 2 transition, if a Node in the Sercos ring has a node number that is greater than the MEIXmpSercosNodeCountMAX, this error will occur.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessagePROCEDURE_DATA_INVALID	
Meaning	The wrong data type is being used to access an IDN field.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MPISercosMessageMST_RECEIVE_ERROR	
Meaning	The wrong data type is being used to access an IDN field.
Possible Causes	This error occurs when the user is trying to read the Min or Max IDN field on an IDN whose data is Variable.
Recommendations	.If you are receiving this error message, then please contact an applications engineer at Motion Engineering, Inc.

MEISercosMessage

```
typedef enum {
    MEISercosMessage    BUFFER_SIZE_ERROR,
} MEISercosMessage;
```

Description

MEISercosMessageBUFFER_SIZE_ERROR	
Meaning	There is no memory allocated for a given SERCOS object.
Possible Causes	When mpiSercosValidate() is called and the firmware does not have any space allocated for SERCOS.
Recommendations	Make sure the controller has been configured for SERCOS.

See Also

MPISercosProcedureAction

MPISercosProcedureAction

```
typedef enum {
    MPISercosProcedureActionINVALID,

    MPISercosProcedureActionCLEAR_AND_START,
    MPISercosProcedureActionSTART,
    MPISercosProcedureActionCANCEL,
    MPISercosProcedureActionSTATUS,
    MPISercosProcedureActionCLEAR_AND_EXECUTE,
    MPISercosProcedureActionEXECUTE,
    MPISercosProcedureActionNONE,

} MPISercosProcedureAction;
```

Description

SercosProcedureAction lists all of the different commands that can be applied to a SERCOS procedure.

See Also

MEISercosServiceContainer

MEISercosServiceContainer

```
typedef struct MEISercosServiceContainer {
    unsigned long       header[5];

    unsigned long       write[20];
    unsigned long       read[20];
    unsigned long       readOverhang;
} MEISercosServiceContainer;
```

Description **SercosServiceContainer** is part of the Sercos message where noncyclic data is requested and sent. It resides in both the AT and MDT.

See Also For a definition of the individual fields, consult a SERCOS specification manual.

MPISercosStatus

MPISercosStatus

```
typedef struct MPISercosStatus {
    MPISercosLoopStatus  loopStatus;
    long                 phase;
    long                 nodeCount;
} MPISercosStatus;
```

Description

SercosStatus gives the status of the SERCOS ring.

loopStatus	the current state of the ring (ie open or closed)
phase	the phase in which the SERCOS ring is in [0-4]
nodeCount	the number of nodes on the SERCOS ring

See Also

MPISercosNODE_COUNT_MAX

MPISercosNODE_COUNT_MAX

```
#define  MPISercosNODE_COUNT_MAX    (32)
```

Description

SercosNODE_COUNT_MAX defines the maximum number of Nodes that can be on a SERCOS ring.

See Also

mpiSercosNodeIdnDataSET

Declaration

```
#define   mpiSercosNodeIdnDataSET(sercos,node,idn)
          mpiSercosNodeIdnFieldSet((sercos), (node), (idn), MPIIdnFieldDATA)
```

Required Header

stdmpi.h

Description

SercosNodeIdnDataSET writes the data fields from an idn object to an ***idn*** on a sercos ***node***.

See Also

mpiSercosNodeIdnGET

Declaration

```
#define   mpiSercosNodeIdnGET(sercos,node,idn)
          mpiSercosNodeIdnFieldGet((sercos), (node), (idn), MPIIdnFieldALL)
```

Required Header

stdmpi.h

Description

SerocsNodeIdnGET reads all the fields from ***idn*** located on a sercos ***node*** and writes them into the idn object.

See Also

Sercos- Introduction

About SERCOS | Supported Drives

About the MEI SERCOS Controller

The SERCOS/XMP Series controllers from MEI are an extension of the XMP Series motion controllers. XMP Series motion controllers support analog motion control outputs, encoder inputs, and discrete digital I/O. SERCOS/XMP Series motion controllers replace these signals with the SERCOS digital fiber optic network interface. The SERCOS interface only requires two fiber optic connections (one output and one input) to connect to a fiber loop containing up to 8 axes of motion.

Both SERCOS and standard XMP Series controllers share the same basic hardware architecture, onboard firmware, host software and many other features. So, for controller installation procedures, Motion Console application and C programming information, use the XMP's standard documentation.

However, because the SERCOS IDNs are actually implemented in the drive (and not in the controller), there are many SERCOS functions not documented in the XMP documentation, because they are in the drive's documentation (because these functions are associated with the drive and not the controller). Motion Engineering adheres to the specifications set forth by the IEC concerning SERCOS. For more information regarding SERCOS or the SERCOS specification, please contact SERCOS N.A at www.sercos.com.

TOP

Supported Drives / Modules

MEI currently supports drive and I/O modules from a variety of manufacturers. If you desire support for a drive or I/O vendor not listed, please contact MEI.

Manufacturer	Device
Indramat	Servo Drives
Modicon	Servo Drives
Lutze	Digital I/O Modules
Pacific Scientific	Servo Drives
Kollmorgen	Servo Drives
Sanyo Denki	Servo Drives

TOP

Sercos- Overview

Summary | Operation Modes | Closed-Loop Tuning | Data Transmission

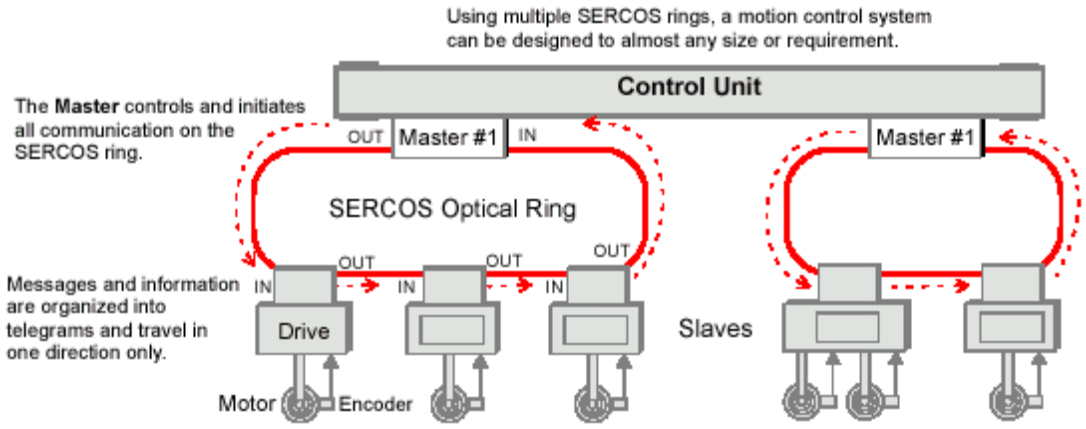
Summary

SERCOS (SErial Real-time COmmunication System) is the international standard for optical communication between motion control units and drive modules. It was developed by the International Electrotechnical Commission (IEC) specifically for motion control and is defined by the IEC 1491 standard. SERCOS supports data rates up to 16 Mbits/sec over a fiber optic ring. Data can be transmitted deterministically in real time based on the loop update rate (cyclic), or at lower bandwidths for less critical operations (non-cyclic). A SERCOS-compatible communication ring must have a single controller (master) and 1 - 254 drive or I/O modules (slaves).

Also, data communication can be performed in either a synchronous or asynchronous manner. The protocol allows the user to configure the communication telegrams to send whatever data is appropriate synchronously. Data that has not been configured to reside in the communication telegrams can be sent or retrieved asynchronously by use of a Service Channel contained within the communication telegrams. Generally, synchronous data is data that is critical to real-time operation (e.g., command and feedback data, status).

SERCOS is a unidirectional serial communications protocol for connecting multiple drives (Slaves) to a motion controller (Master) over a fiber optic ring, in an industrial environment. The control and status information is organized into telegrams, and travels in a serial data stream around the SERCOS ring. All messages are synchronized according to the SERCOS cycle time, the timing of which is configured during initialization by the Master. Each Slave on the ring repeats the telegrams sent to it, sending them to the next device on the ring, and inserting its own telegram into the designated time slot in the serial data stream.

Starting at the output port of the Master, the devices are connected in the ring in a daisy-chain fashion, connecting from the output port of one device to the input port of the next device, and so on, until the ring is closed back at the input port of the Master. Up to 254 drives can be connected on a SERCOS ring, although the systems requirements for update rates and data will usually limit the number of devices to many fewer than that. SERCOS networks can operate at 2, 4, 8 or 16 Mbit/sec. Maximum distances from input port to output port can be 60 meters (plastic fiber) to 250 meters (glass fiber).



TOP

Operation Modes

The SERCOS communication interface supports three main operation modes (Torque, Velocity and Position). The operation mode defines the real-time digital messages sent between the controller and the drive(s).

Mode	To Drive	To Controller
Torque	16 bit Command Torque	32 bit Actual Position
Velocity	32 bit Command Velocity	32 bit Actual Position
Position	32 bit Command Position	32 bit Actual Position

In addition to the main operation modes SERCOS supports several variations. Since the communication interface is determined by the firmware/software in the controller and the firmware/software in the drive, the real-time data is configurable.

Currently the SERCOS/XMP Series firmware/software supports several operation modes. Some operation modes are drive specific while others are drive independent. Motion Engineering is constantly testing and certifying compatibility between our controller and SERCOS-compatible drives.

Most drives support the three main operation modes (Torque, Velocity and Position). Please consult your drive specific documentation regarding supported operation modes and variations.

In all modes, the controller calculates a 32-bit command position every sample. The command position is based on the current command jerk, command acceleration and command velocity.

In Torque mode, the controller sends a 16-bit command torque to the drive. The drive sends a 32-bit actual position back to the controller. Every sample, the controller calculates a new command torque based on the position error and the control algorithm. The controller closes the position and velocity loop and the drive closes the torque loop.

In Velocity mode, the controller sends a 32-bit command velocity to the drive. The drive sends a 32-bit actual position back to the controller. Every sample, the controller calculates a new command velocity based on the position error and the control algorithm. The controller closes the position loop, and the drive closes the velocity and torque loop.

In Position mode, the controller sends a command position to the drive. The drive sends a 32-bit actual position back to the controller. Every sample, the controller calculates a new command position. The drive closes the position, velocity and torque loop.

TOP

Closed-Loop Tuning

A general difference between SERCOS digital drives and conventional analog drives is that digital drives have on-board intelligence and can close position or velocity loops within the drive. In all operation modes, “tuning” requires setting parameters in the drive and the controller. Thus, an understanding of both the controller and drive control algorithm is necessary for successful drive tuning.

The controller tuning parameters can be set using Motion Console (for Windows-based systems). For more information on the controller’s tuning procedures, please consult the Tuning section in the *XMP Motion Controller Hardware Installation* manual.

The drive’s tuning parameters must be set via IDNs, and the values are determined from information supplied by the drive manufacturer. Please consult the drive-specific documentation for more information on the drive’s tuning parameters. While Motion Engineering has considerable experience with the listed drives and can generally offer tuning guidelines for drive parameters, difficult tuning situations may require support from the drive vendor or manufacturer.

In Velocity mode, the controller's CONTROL ALGORITHM output controls the motor's velocity. Therefore, the drive's velocity loop must be tuned and the controller's position loop must be tuned. The tuning procedure is identical to a standard analog output XMP Series controller connected to a velocity-controlled amplifier. Note that the controller's Velocity Feed Forward term is very useful in velocity-controlled systems.

In Position mode, the controller's control algorithm is not used. The drive is responsible for the closed loop control. Therefore, the drive's velocity and position loop must be tuned. The controller's tuning parameters have no effect on the system's response.

TOP

Data Transmission

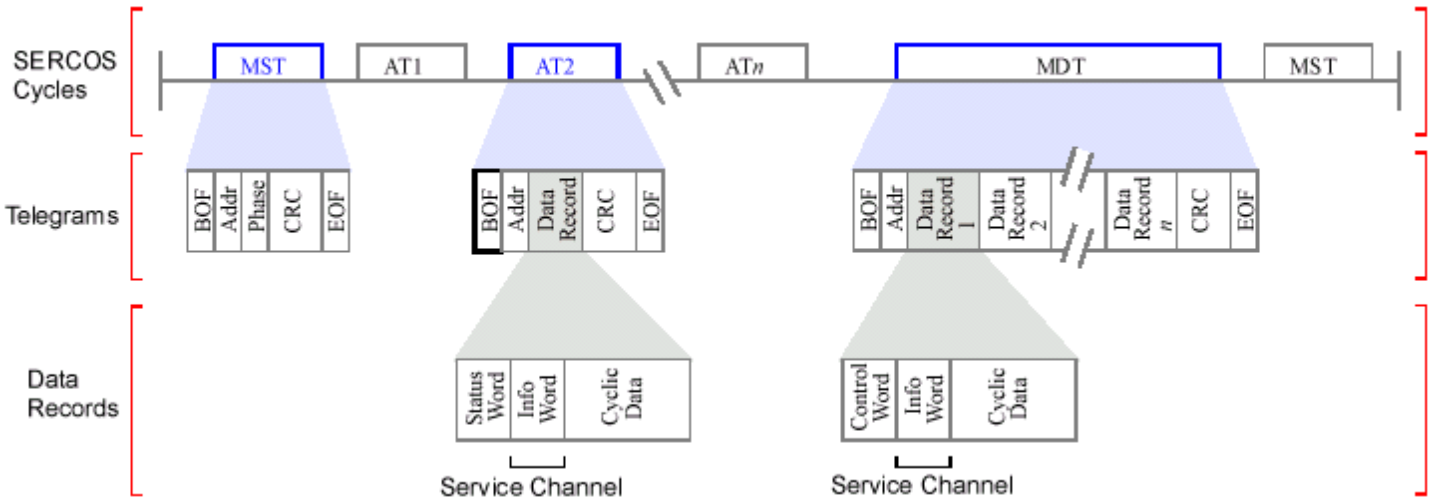
SERCOS supports two types of data transmission, cyclic and non-cyclic.

Cyclic data is the critical real-time synchronized data sent between the Master and the Slaves (drives, I/O modules). In every SERCOS cycle, the Master sends and receives fixed-length messages to the drives and I/O modules. These messages contain the motion control command signal and the feedback response for each drive or the digital I/O commands and responses for each I/O module. The cyclic data is guaranteed to reach each drive and I/O module and return to the Master at a fixed time interval, the SERCOS cycle.

Non-cyclic data is the noncritical asynchronous data. The cyclic fixed-length messages have space (called the Service Channel) reserved for non-cyclic data. In each SERCOS cycle, the Master may transmit two bytes of non-cyclic data through the Service Channel to each Slave. Note that it may require several SERCOS cycles for the Master to complete the transmission of the non-cyclic data to the Slaves. Typically, transmitting non-cyclic data is much slower than cyclic data.

SERCOS cycles are built using telegrams, which in turn contain data records for all of the Slave drives. Cyclic data is transferred in the cyclic data part of the data records. Non-cyclic data is transferred in the Service Channel of data records. Refer to the next figure.

TOP



Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

Sercos- Data Types

Summary

And datas in a slave that can be accessed by the host is represented by a data block. Data blocks are assigned identification numbers (IDNs or “ID numbers”). The data block consists of information about the data; for example, IDNumber, name, attributes, units, min and max values, and the operation data itself.

TOP

Data Block Structure

The SERCOS protocol is designed to handle many different types of data, characterized by two fundamental types: fixed length data and variable length data.

Fixed length data is either 2 bytes or 4 bytes wide and can be used to represent signed or unsigned integers, hexadecimal values, binary codes, IDNumbers (identification numbers) of other Data Blocks, and procedure commands.

The length of **variable length** data depends on what type of data is present, and is defined by the first two words (32 bits), which specify the actual and maximum length of the data. Variable length data can be used to represent character strings, lists of IDNumbers of Data Blocks, lists of signed or unsigned integers (both 2 or 4 bytes wide), lists of hexadecimal values, etc.

All data (fixed and variable length) can be sent or received via the Service Channel. However, **only** fixed length data is allowed to be configured into a communication telegram (MDT, AT). When a communication telegram is configured to send or receive fixed length data, it is only Element 7 data that will be either sent or received.

TOP

Data Block Structure of IDNs

Data is accessed through data blocks referred to as IDNs. An IDN consists of seven elements:

Element 1	IDNumber
Element 2	Name
Element 3	Attribute
Element 4	Unit
Element 5	Minimum Input
Element 6	Maximum Input
Element 7	Operation Date

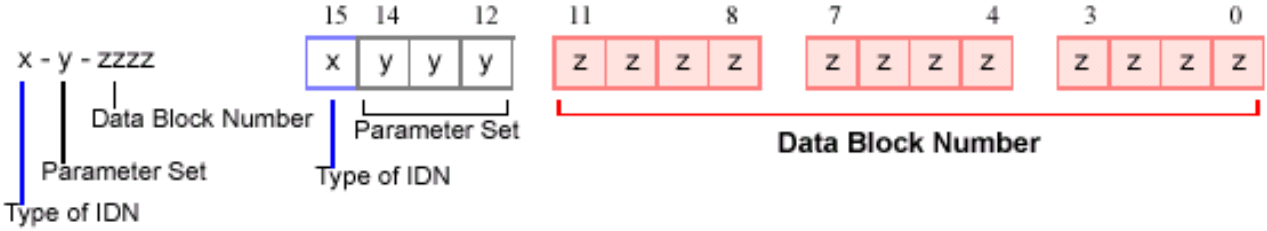
All data exchanged between Master and Slaves has an IDNumber (IDN number) assigned to it. Every IDN has an associated data block which consists of seven elements. The Master can only write Element 7 data; the Master cannot write Elements 1 - 6. Elements 1 - 6 are defined by the drive itself.

Element	Description	Data Type
1	IDNumber (Identification number)	**binary** Expressed as either S-X-XXXX or P-X-XXXX. S denotes IDNs that are defined by the SERCOS Specification. P denotes IDNs that are defined by the manufacturer of the device.
2	Name	**variable length string** Contains the name of the IDN.
3	Attribute	**32 bit binary** Contains information about conversion factors and data representation (signed or unsigned integer, fixed or variable length data, etc.)
4	Unit	**variable length string** Contains a representation of the units for the data.
5	Minimum input value	1 or 2 words
6	Maximum input value	1 or 2 words
7	Operation Data	1 or 2 words, or string

TOP

Element 1: IDNumbers

Each Data Block has a number assigned to it for identification purposes, called the IDNumber. The IDNumber is represented as either S-X-XXXX or P-X-XXXX. S denotes a Data Block that is defined by the SERCOS Specification. P denotes a Data Block that is defined by the manufacturer. The first 'X' identifies the "data set" that the Data Block belongs to. According to the SERCOS Specification, it is possible to switch between data sets.

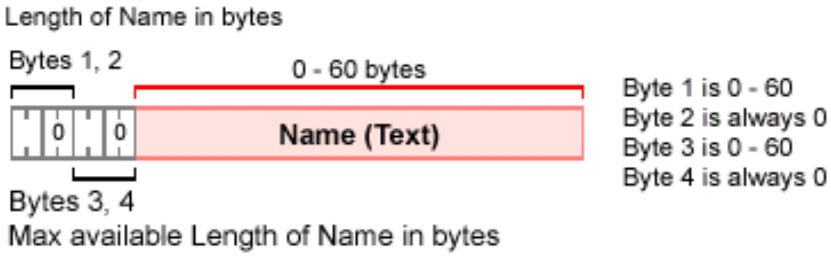


Bits	Name	Values
15	Type of IDN	0 S - Standard data 1 P - Product-specific data as the "S" or "P" part of the IDN notation S-y-zzzz or P-y-zzzz.
14-12	Parameter Set	0 - 7, as the "y" part of the IDN notation S-y-zzzz or P-y-zzzz.
11-0	Data Block Number	0 - 4095, as the "zzzz" part of the IDN notation S-y-zzzz or P-y-zzzz.

TOP

Element 2: Name of Operation Data

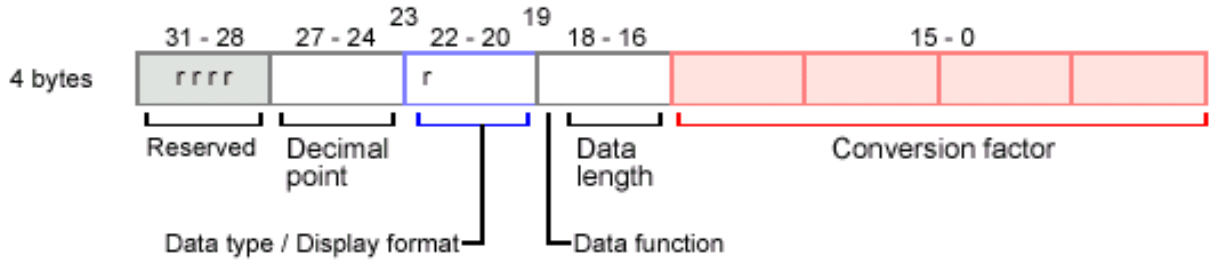
The name of operation data is 64 bytes maximum, with 2 length specifications of 2 bytes each, and a character string of 60 characters maximum. Bytes 1 and 2 contain the number of characters in the text. Bytes 3 and 4 contain the maximum number of characters in the text. Since this element is READ-Only, bytes 3 and 4 will contain the same values as bytes 1 and 2.



TOP

Element 3: Attributes of Operation Data

Every data block has an attribute (4 bytes) which contains all of the information required to display operation data, using universal routines. If data needs to be scaled (to be displayed), then specific scaling parameters are supplied in the attribute.



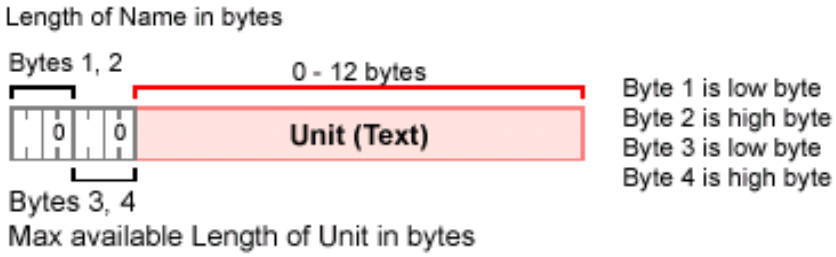
Bits	Name	Values	
31	Reserved		
30	Phase 4 write-protection bit.	1 = write-protected	
29	Phase 3 write-protection bit.	1 = write-protected	
28	Phase 2 write-protection bit.	1 = write-protected	
27-24	Number of places after the decimal point Indicates the position of the decimal point in the data to be displayed. Basically, it's the exponent "x" in 10-x.	000 No places after decimal point 001 1 place after decimal point 010 2 places after decimal point * 111 15 places after decimal point	
23	Reserved		
22-20	Data Type & Display Format Used to convert the operation data, and min/max input values to the correct display format.	**Data Type** 000 binary number 001 unsigned integer 010 integer 011 unsigned integer 100 extended char set 101 unsigned integer 010 Reserved 111 Reserved	**Display Format** binary unsigned decimal signed decimal hexadecimal text IDN number
19	Function of Operation Data Indicates whether this data calls a procedure in a drive	0 Operation data or parameter 1 Procedure command	

18-16	Data Length The data length is required so that the Master is able to complete Service Channel data transfers correctly.	**000 Reserved** 001 Operation data is 2 bytes long 010 Operation data is 4 bytes long **011 Reserved** 100 Variable length with 1-byte data strings 101 Variable length with 2-byte data strings 110 Variable length with 4-byte data strings **111 Reserved**
15-0	Conversion Factor	An unsigned integer used to convert numeric data to display format. Is set to 1 when it is not needed for data display (e.g., binary display or a character string).

TOP

Element 4: Operation Data Unit

The operation data unit is 16 bytes maximum, with 2 length specifications of 2 bytes each, and a character string of 12 characters maximum. Bytes 1 and 2 contain the number of characters in the text. Bytes 3 and 4 contain the maximum number of characters in the text. Since this element is READ-Only, bytes 3 and 4 will contain the same values as bytes 1 and 2.



TOP

Element 5: Minimum Input Value of Operation Data

The minimum input value is the smallest numerical value for operation data that the drive can process. When the Master writes a value to the drive that is less than the minimum value, the drive ignores it and continues to use the previous operation data.

When the operation data is of variable length or a binary number is used, there is no minimum input value of operation data.

TOP

Element 6: Maximum Input Value of Operation Data

The maximum input value is the largest numerical value for operation data that the drive can process. When the Master writes a value to the drive that is more than the maximum value, the drive ignores it and continues to use the previous operation data.

When the operation data is of variable length or a binary number is used, there is no maximum input value of operation data.

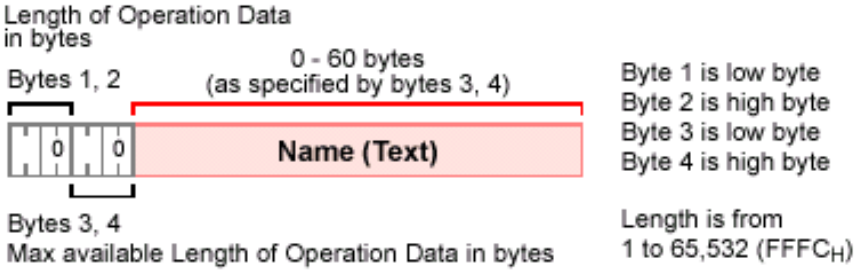
TOP

Element 7: Operation Data

In terms of length, there are 3 types of operation data:

- fixed length with 2 bytes
- fixed length with 4 bytes
- variable length up to 65,532 bytes in 1 byte (char), 2 byte, or 4 byte values

Bytes 1 and 2 contain the number of bytes in the text. Bytes 3 and 4 contain the max number of bytes available in the text. Data in the text may be 1 byte, 2 bytes, or 4 bytes wide.



TOP

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

NEXT

Sercos- Communications

Synchronization | Ring Timing | Initialization

Synchronization

The Master is responsible for sending a synchronization telegram (MST) at the beginning of each communication cycle. All Slaves (drives) will receive the MST and reset their clocks. In this way, all Slaves will run in phase lock with the Master's clock. Because all Slaves are in phase lock with the Master's clock, commands can be made active in all of the Slaves at the same instant. This means that the Master can coordinate motion between all axes without propagation effects distorting the motion profile. Feedback from the Slaves is handled in a similar manner, and is latched in all Slaves at the same instant.

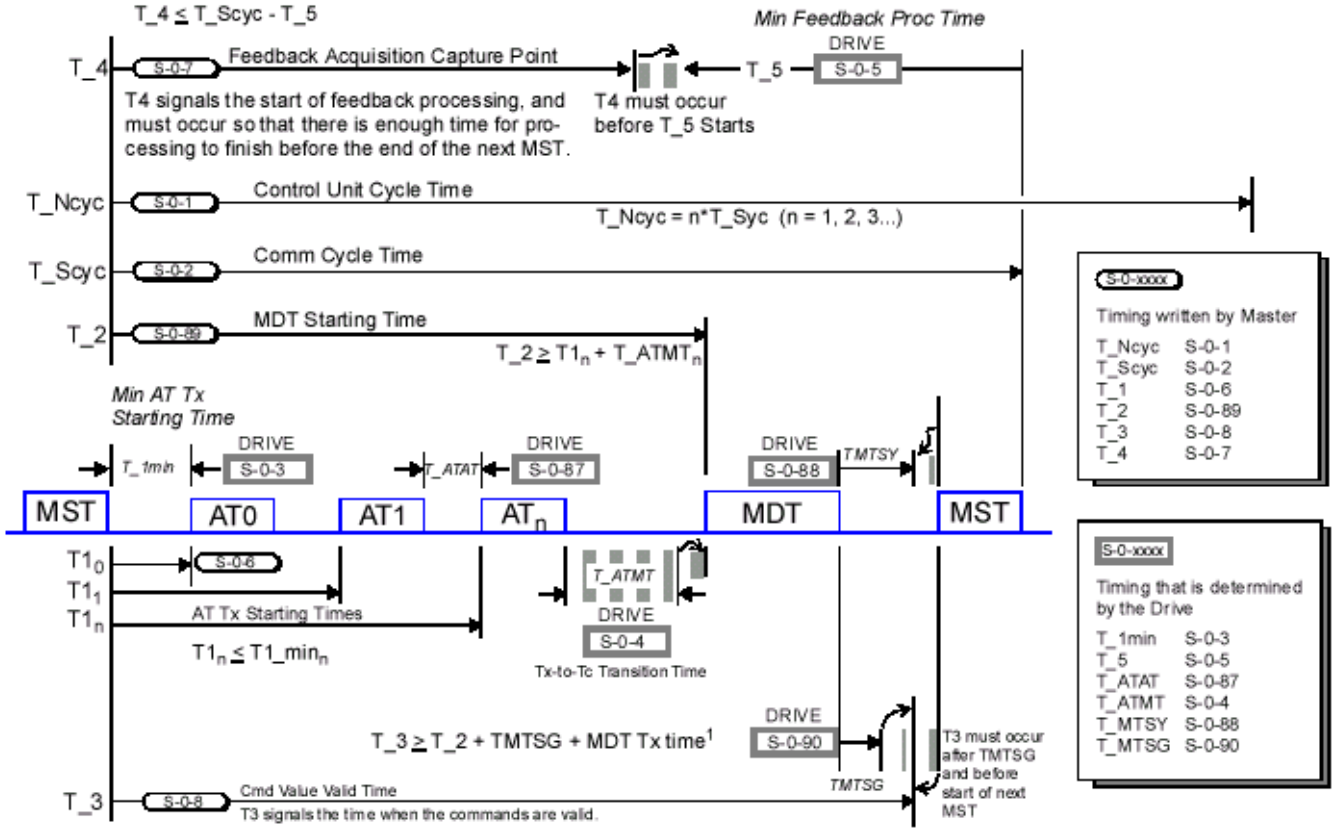
TOP

Ring Timing

Data is sent and received by the Master and Slaves through communication telegrams. The communication telegrams are organized over the SERCOS cycle in the manner shown in the next figure.

The SERCOS ring timing is based on the data to be placed in the telegrams (AT, MDT), and on 6 drive parameters that are determined by the type and features of a drive (or drives), and 6 parameters that are written from the Master, with some of these parameters derived or calculated from the drive's timing parameters. The times at which the MDT and AT are sent (relative to the sending of the MST) are determined by the Master and sent to the Slaves during initialization.

During Phase 2, the Master reads parameters from the drives that determine what and when the drives are able to transmit and receive. Using this information and the desired telegram contents, baud rate and cycle time, the Master determines the timing and telegram parameters for each drive. The Master then writes these parameters to the drives.



[1] *MDT Tx time* depends on the number of slaves and the amount of data sent to the slaves. The Master must calculate the *MDT Tx time* during Phase 2.

For a print-friendly version of the above diagram, click here.

TOP

Initialization

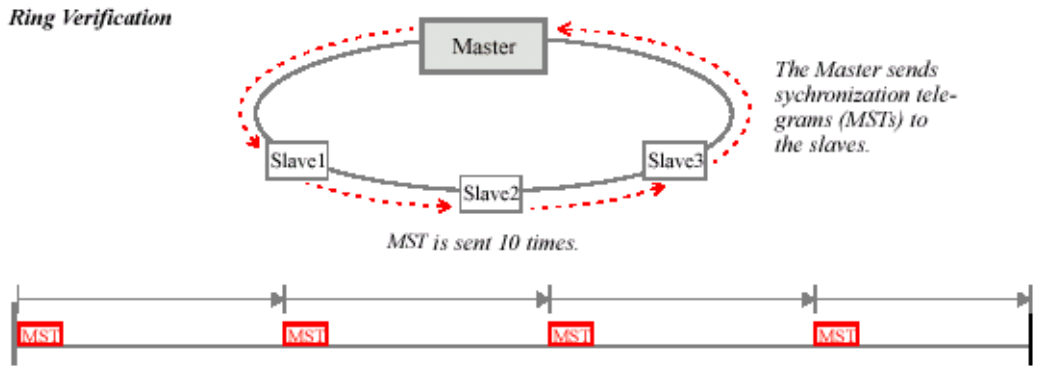
Before true synchronous data transmission can occur, the system must first be initialized. This is done through a series of **communication phases** (or just phases) in which data is first transmitted asynchronously. The data transmitted during these early phases is used to configure the Master and Slaves for synchronous data transmissions in later phases. The SERCOS protocol defines five phases.

Phase	Name	Action
0	Ring Verification	Master verifies ring closure
1	Device Verification	Master verifies devices on ring
2	Telegram Set-Up	Master reads timing data from slaves and sets up telegram timing
3	Device Parameterization	Master continues to configure devices
4	Cyclic Operations	Master commands devices cyclically

On power-up, each drive or I/O module begins an initialization sequence. At this time, each drive and I/O module operates as a repeater, by simply passing received telegrams to the next device on the SERCOS ring.

The Master is only allowed to set the communication phase to the next logical communication Phase **(Phase 0 -> Phase 1 -> Phase 2 -> Phase 3 -> Phase 4)** or directly back to **Phase 0**. If at any time the Master attempts to switch a Slave into a Phase that is not the next logical Phase, then the Slave will immediately return to Phase 0. If at any time the Slave receives two invalid MSTs or MDTs consecutively, the Slave will also switch to Phase 0.

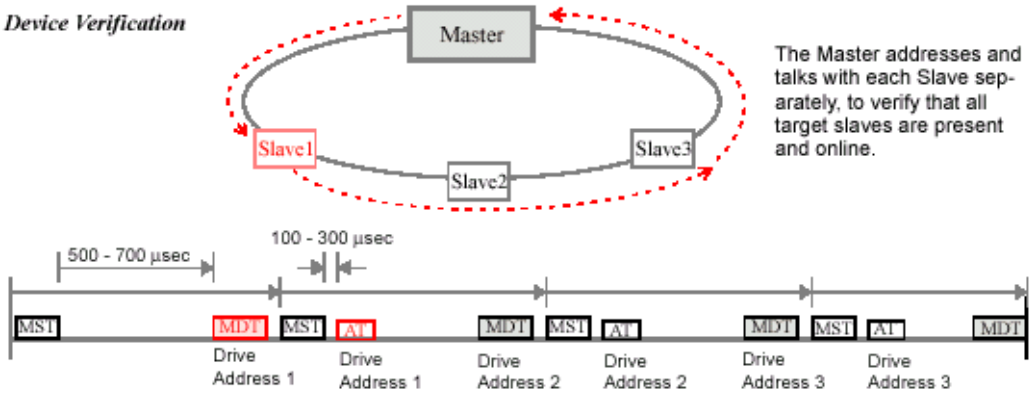
Phase 0:



During **Phase 0** **no data is exchanged** between the Master and the Slaves. All Slaves must be in “repeater” mode. This means that each Slave will retransmit any signal that it receives. In order to verify that the communications ring is intact and capable of sending telegrams, the Master begins sending Master Synchronization Telegrams (MST) through the SERCOS ring.

The Slaves (drives) simply pass the MST to the next drive in the daisy-chained ring, and eventually because of the ring topology, the MST returns to the Master (i.e., Master will receive its own MST). Phase 0 is completed when the Master receives 10 consecutive MSTs. After the tenth consecutive MST, the Master changes the phase information in the MST to 1, which commands all Slaves to switch into Phase 1 operations.

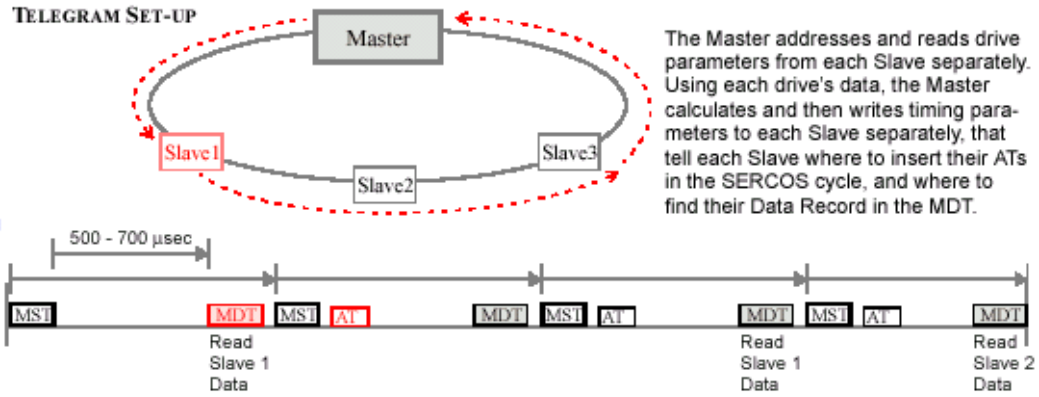
Phase 1:



During **Phase 1**, the Master sends out an MDT with the address of a specific Slave in the system. If present within the system, the Slave with the specified address responds by sending an AT back to the Master. The AT that is sent is rudimentary and is intended solely as a confirmation that the addressed Slave is in the system. The Master then repeats this query for all target Slaves. (Note that not all Slaves in the system will be target Slaves. Target Slaves are chosen by the application before initialization begins.)

When the Master receives an AT from each target Slave (in response to a query), the Master changes the phase information in the MST to 2, which commands all Slaves to switch into Phase 2 operations.

Phase 2:



During **Phase 2**, the Master sets up the configurable data portion and calculates the duration and time slots within the SERCOS cycle for all telegrams to be used in Phases 3 and 4. The Master also determines the slave operation mode for all Slaves. In order to do this, the Master requires certain data from the Slaves. The Master obtains this data by sending an MDT addressed to a specific Slave, that uses the MDT's Service Channel to query the Slave for the required data

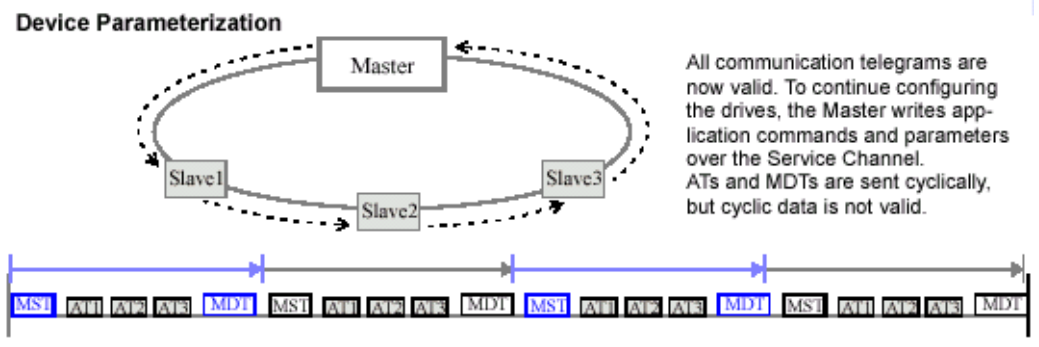
The Slave responds by sending an AT containing the appropriate data, in the AT's Service Channel. Once the Master has determined all parameters, it sends them to each Slave via the MDT's Service Channel. In Phase 2, the Service Channel is active in both MDTs and ATs.

Once all data is transmitted to the Slaves, the Master will initiate the Communication Phase 3 Transition Check S-0-127 procedure for each Slave.

The Communication Phase 3 Transition Check checks the validity of all the data, and if all the data is valid, then the procedure executes successfully. If any data is not valid, the procedure fails and the IDN number of the invalid data is placed in IDN-List of Invalid Operation Data for CP2 S-0-21.

After all Slaves have completed the procedure successfully, the Master changes the phase information in the MST to 3, which commands all Slaves to switch into communication Phase 3.

Phase 3:



In **Phase 3**, the real-time synchronous and asynchronous communication starts, and the Master uses the Service Channel to configure and parameterize (write parameters to) the Slaves. The parameters set in Phase 3 are application-oriented (e.g., conversion factors). During Phase 3, all communication telegram parameters sent in Phase 2 become active. Note that although Configurable Data (in ATs and MDTs) is present in the communication telegrams, some of that Configurable Data may not be valid until Phase 4.

The Master will still send the MST at the beginning of the SERCOS cycle, but will now also send an MDT with a global address at a specific time (all Slaves will receive a global telegram). Each Slave will transmit its AT during its specified time slot. The communication telegrams now contain the Control/Status Word, a Service Channel, and the Configurable Data.

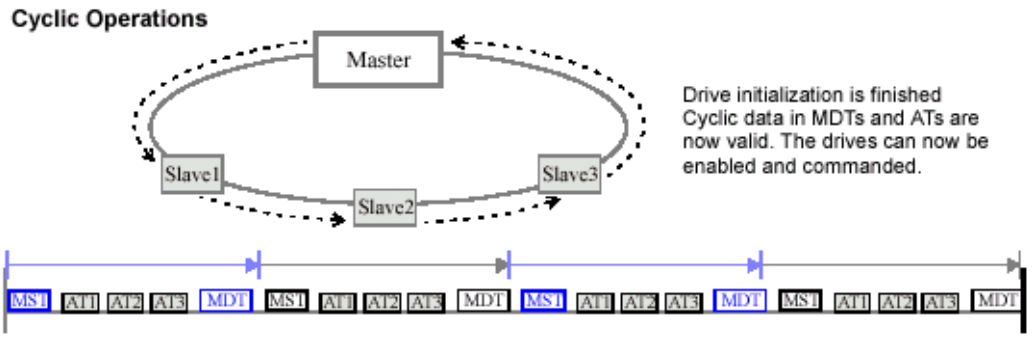
When the Master has finished sending parameters to the Slaves, it will initiate the Communication Phase 4 Transition Check S-0-128 procedure for each Slave.

The Communication Phase 4 Transition Check checks the validity of all the data, and if all the data is valid, then the procedure executes successfully. If any data is not valid, the procedure fails and the IDN number of the invalid data is placed in IDN-List of Invalid Operation Data for CP3 S-0-22.

TOP

If the Communication Phase 4 Transition Check executes successfully and the Phase information in the MST is equal to 4, the Master switches the drives to Phase 4.

Phase 4:



In **Phase 4**, a final verification of error-free drive operation is completed. This completes the drive initialization. The SERCOS communication loop is now operational. During Phase 4, all Control/Status words, Service Channels, and Configurable Data (in ATs and MDTs) are valid for the target Slaves. The Slaves (drives) are ready to follow commands when enabled. Diagnostics (errors, warnings, status) are enabled.

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

NEXT

Sercos- Procedures

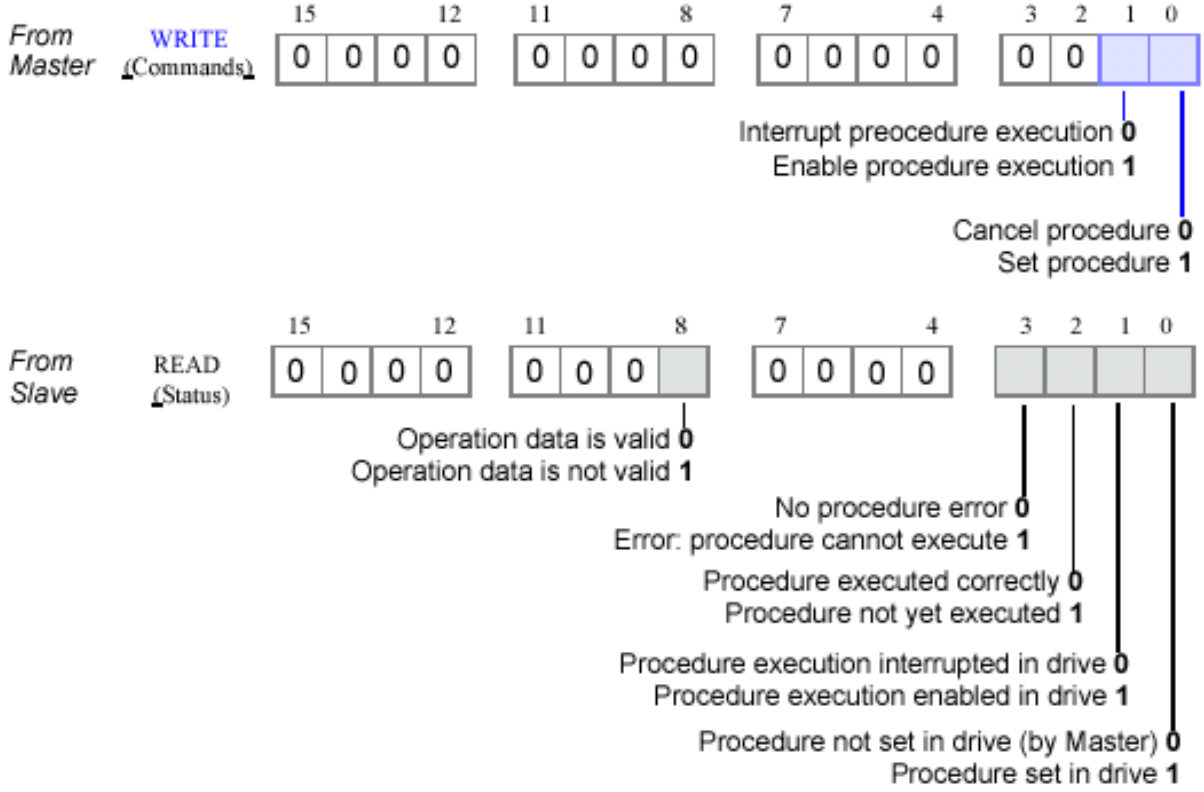
Summary

Many Slaves (drives) come with preprogrammed procedures that the Slave is able to execute without assistance from the Master. Each of these procedures is assigned a Data Block. The Master controls a procedure by reading and writing to Element 7 of the procedure's Data Block. The Master can initiate, interrupt, or cancel a procedure at any time by setting or clearing bits in Element 7 of the procedure's Data Block.

Bits 0 and 1 are responsible for respectively setting and enabling the procedure command. Once set and enabled, the Slave will begin execution of the procedure. To indicate that the procedure is executing, the Slave sets bit 2 of the procedure status word. When the procedure has finished executing, the Slave clears bit 2 of the procedure status word and sets the Procedure Command Change Bit in the Drive Status Word S-0-134. If an error has occurred when executing the procedure, the Slave will set bit 3 of the procedure status word and will also set the Procedure Command Change Bit in the Drive Status Word S-0-135. In either case (successful execution or failure), the Master must cancel the procedure by clearing bits 0 and 1 of the procedure.

In SERCOS, in order for a Master to execute a procedure,

1. The Master writes 0x3 to Element 7 of the desired IDN procedure, which sets and enables the procedure.
2. The Master reads the procedure status word and checks bits 2 & 3 to see if the procedure has finished executing or if there is an error.
3. After reading that the procedure has executed, the Master writes 0x0 to Element 7 of the particular procedure, which cancels the procedure.



TOP

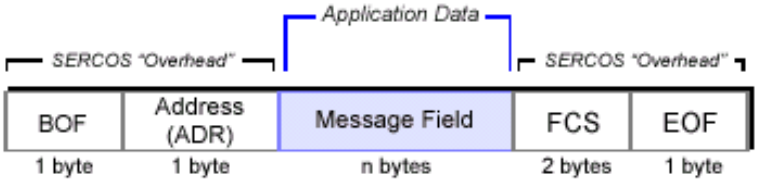
Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

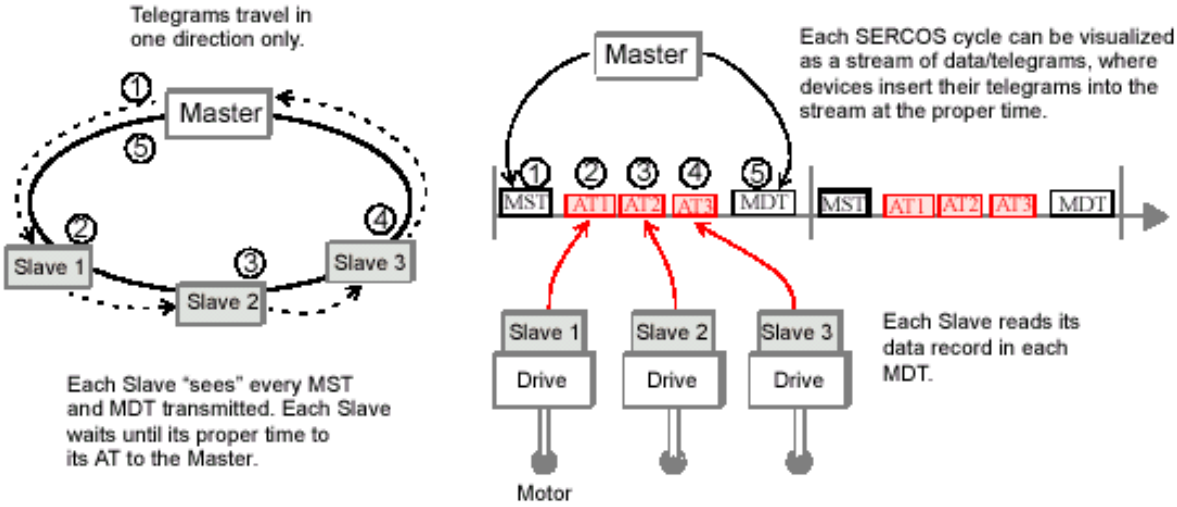
Sercos- Telegrams

Summary

The controller (Master) communicates with the drives and I/O modules (Slaves) using telegrams. A telegram is a structure that contains data, error checking and handshaking information. SERCOS supports three types of telegrams: Master Synchronization Telegram (MST), Master Data Telegram (MDT), and Amplifier Telegram (AT).

Each type of telegram contains 5 types of fields: BOF (beginning of frame), ADR (address), message field, FCS (frame check sequence), EOF (end of frame).





TOP

BOF Delimiter

All telegrams have a BOF (beginning of frame) byte denoting the beginning of the telegram. The BOF is always 0111 1110.

TOP

ADR Target Addresses

All telegrams have an ADR (address) byte, which denotes the address of a drive (Slave). In a telegram from the Master, the address specifies which drive the information is for. In a telegram from a Slave (drive), the address specifies the sourcing drive. The target addresses for the drives are valid if greater than 0 and less than 255. Typically, a drive's address is set using a selector located physically on the drive.

Address 0 is the "no station" address, and is sometimes used to remove a drive from the ring logically, during troubleshooting. During non-cyclic operations (Phases 0 - 2), the Master can only communicate with one drive per cycle. During cyclic operations (Phases 3, 4), the Master can communicate with all drives.

From	**To**	**Telegram**	**Non-Cyclic** (Phases 0,1,2)	**Cyclic** (Phases3,4)
Master	Slave	**MST**	255	255
Master	Slave	**MDT**	$1 \leq ADR \leq 255$	255
Slave	Master	**AT**	$1 \leq ADR \leq 254$	$1 \leq ADR \leq 254$

TOP

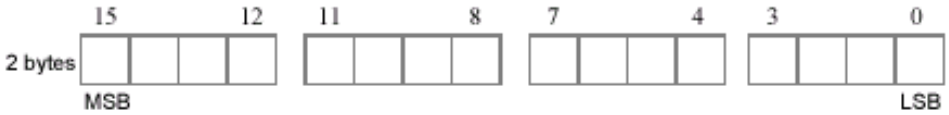
Message Field

Each Telegram (MST, AT, and MDT) contains a message field. The Message Field for the MST consists of one 8-bit word. The lower three bits of the MST contain Phase information. The Message Field for the AT consists of one Data Record. The Message Field for the MDT consists of one Data Record if the MDT's target address is a specific drive. If the MDT's Target address is all drives (255), then the Message Field for the MDT consists of one Data Record per drive in the system.

TOP

FCS (Frame Check Sequence)

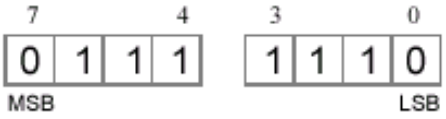
All telegrams have a two-byte FCS number used to check data integrity. The frame check sequence (16 bits) is implemented according to ISO/IEC 3309, 4.5.2.



TOP

EOF (End of Frame) Delimiter

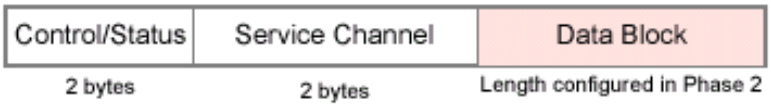
All telegrams have an EOF byte denoting the end of the telegram. The EOF is always 0111 1110.



TOP

Data Record

Data Records are used by both the Amplifier Telegram (AT), and the Master Data Telegram (MDT) to send data. Generally, a Data Record consists of a 16-bit Control/Status word, a 16-bit Service Channel, and a Data Block of 16-bit words.



Amplifier Telegrams contain only one Data Record, because each Slave sends its own Amplifier Telegram.

During Phase 2, the Master sends a Master Data Telegram that is addressed to a specific Slave. Since only one Slave is to receive the MDT, the Phase 2 MDT has only one Data Record. During Phases 3 and 4, the Master sends a Master Data Telegram that is received by all Slaves on the ring. Since all Slaves receive the Master Data Telegram, the Phase 3-4 MDT has one Data Record for each Slave.

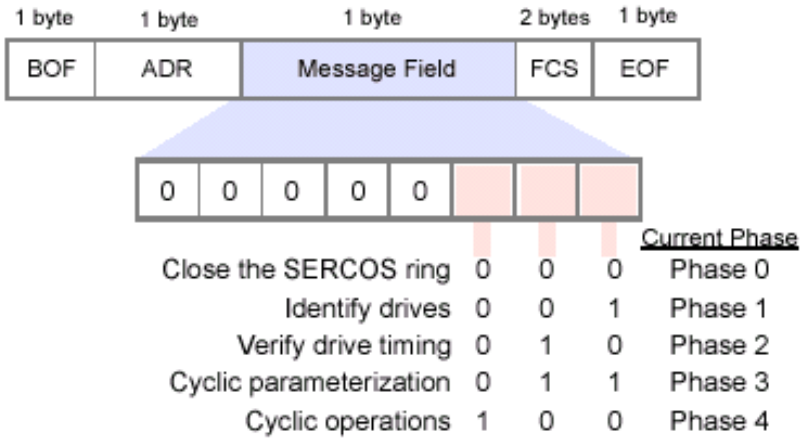
During Phase 2, the Data Block in both the AT and the MDT has a length of 0. This means that all data exchange between Master and Slave must take place using the Service Channel. Because the Service Channel is only 16-bits wide, data exchange can take multiple cycles. Service Channel exchange of data is referred to as **non-cyclic**.

During Phase 2, the Master is responsible for determining the data fields for the Data Blocks in the Phase 3-4 AT and MDT. The size and number of data fields will determine the size of the Data Block. Once defined in Phase 2, the fields and therefore the size of the Data Block is fixed. The data fields in the MDT usually contain command data. The data fields in the AT usually contain feedback data. Because the Data Blocks are configured in Phase 2, there is no overhead using the Data Blocks to send data in Phases 3 and 4. Data exchange using the Data Blocks is referred to as **cyclic**.

TOP

Master Synchronization Telegram

The Master Controller uses master synchronization telegrams (MST) to coordinate its transmission cycle timing with the Slaves. The Master initiates a SERCOS cycle by transmitting an MST to all of the drives and I/O modules on the ring.The MST message field contains one 8-bit word, of which the three lowest bits determine the communication phase of the system. The MST is sent at the beginning of the SERCOS cycle in all communication phases.



Field	Bytes	Description
BOF ADR	1 1	Beginning of Frame. The BOF marks the start of a telegram. The target address. In the MST, ADR = 255 (the broadcast address) during Phases 3 and 4.
Current Phase	1	The lower 3 bits designate the SERCOS phase (0 - 4)
FCS EOF	2 1	Frame check sequence. The FCS field contains circular redundancy check (CRC) information. End of Frame. The EOF marks the end of the telegram.

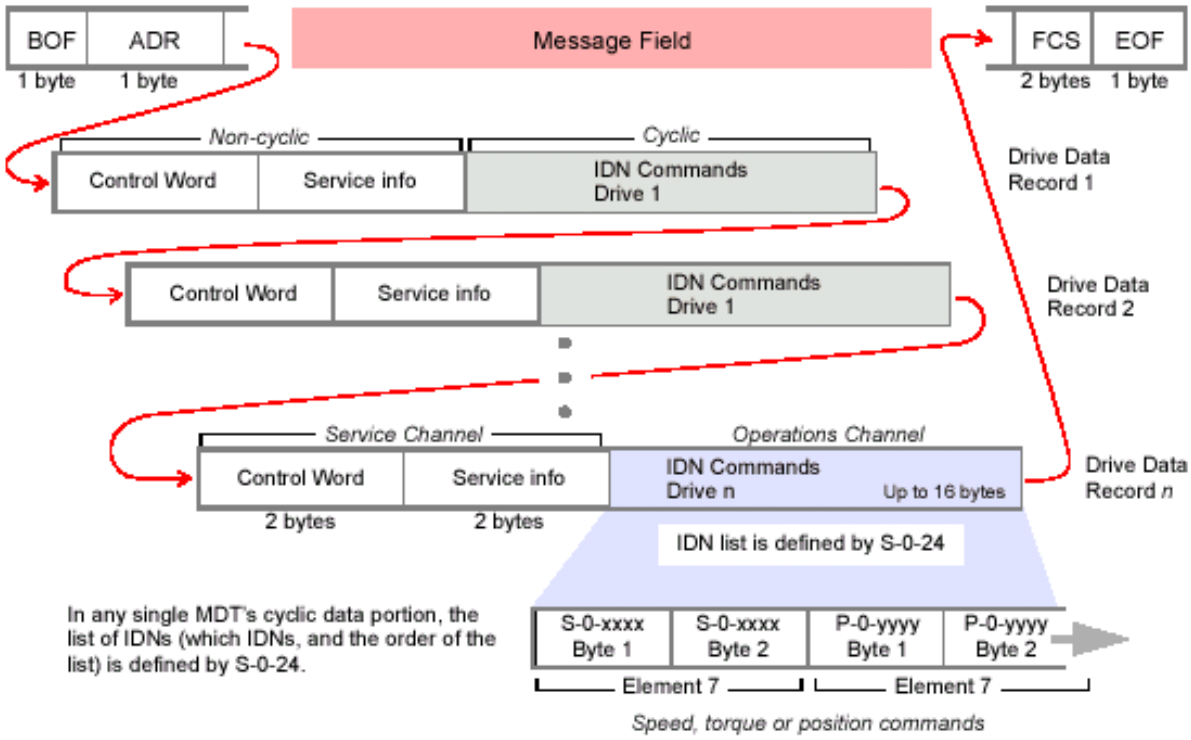
TOP

Master Data Telegram

During Phase 2, the Master must communicate with the Slaves in order to configure them for operations in Phases 3 and 4. In order to send or request data, the Master will send a Master Data Telegram (MDT) to a specific Slave (the Slave is addressed explicitly). Since only one Slave is to receive the MDT, the MDT Message Field contains only one Data Record, where the length of the Data Block (inside the Data Record) is 0, i.e., the data block is empty. During Phase 2, the Master sends data to each Slave via the Service Channel.

During Phase 2, the Master informs the Slave of the byte offset into the MDT at which the Data Record resides for that Slave. The length (in bytes) of the Data Block within each Data Record depends on the data fields configured by the Master during Phase 2. The data fields within the Data Block usually contain command information.

During Phases 3 and 4, once per SERCOS cycle the Master sends a Master Data Telegram (MDT) that has a **global** (broadcast) address. Because the global address is used, all Slaves receive the MDT. Since all Slaves will receive the MDT, the MDT Message Field contains one Data Record for each Slave in the system. During Phases 3 and 4, data is sent to a Slave by using the Data Blocks (cyclic) or the Service Channel (non-cyclic).



Field	Size (bytes)	Description
BOF	1	Beginning of Frame, which is always 0111 1110. The BOF marks the start of a telegram.
ADR	1	Target address. In the MDT, ADR = 255 (the broadcast address).
*Control Word	2	Control word for drive n. Contains operational data.
*Service Info	2	Contains the non-cyclic data for drive n.
*IDN Commands	Variable **	Contains the cyclic data for drive n.
FCS	2	Frame check sequence. Contains circular redundancy check (CRC) info.
EOF	1	End of Frame, which is always 0111 1110. The EOF marks the end of the telegram.

* These fields comprise the data record. There is one data record per drive in the MDT.
** The length if the IDN Commands field is determined during Phase 2

Bit	Name & Value	More Detail
15	0 Drive OFF 1 Drive ON	Bit 15-13=111, the drive should follow command values. When 1->0, the drive removes torque from the motor, and allows the motor to spin down.
14	0 Drive Disable 1 Drive Enable	When 1->0, torque is immediately disabled, independent of bits 15 and 13.
13	0 Drive Halt 1 Drive Restart	
12, 11	**Reserved**	
10	Control Unit Synchronization Bit	

9, 8	Operation Mode 00 primary op mode 01 secondary op mode 1 10 secondary op mode 2 11 secondary op mode 3	Defined byS-0-32 Defined by S-0-33 Defined by S-0-34 Defined by S-0-35
7	Real-time Control Bit 2	S-0-302
6	Real-time Control Bit 1	S-0-300
5, 4, 3	Data Block Element 000 Service channel not active, 001 IDN (number) of the op data 010 Name of operation data 011 Attribute of op data 100 Units of op data 101 Min input value 110 Max input value 111 Operation data	-Close service channel or break a transmission in progress -The service channel is closed for the previous IDN and opened for a new IDN.
2	0 Transmission in progress 1 Last transmission	
1	0 Read service info 1 Write service info	
0	Master Service Transport Handshake	A toggle bit

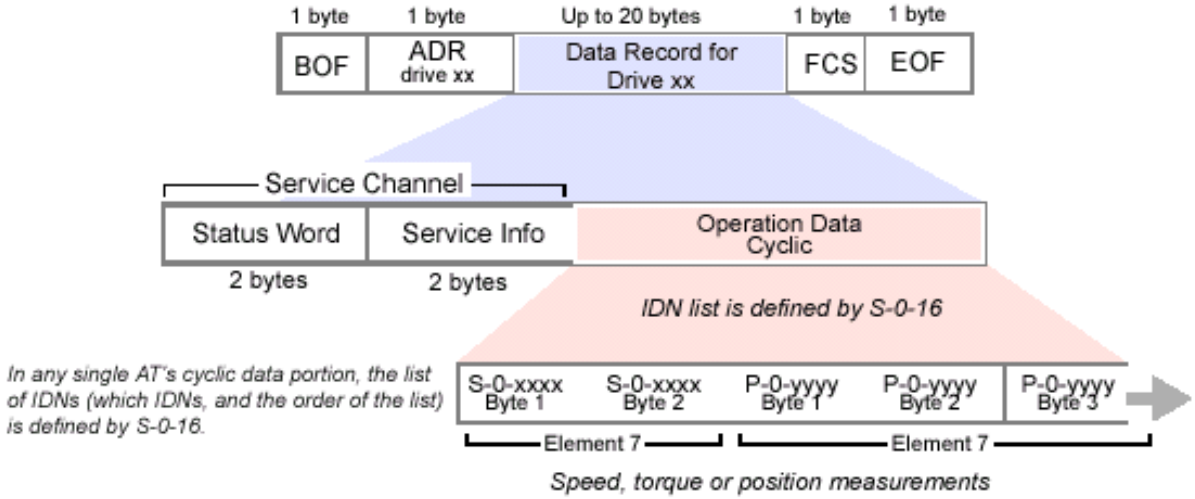
TOP

Amplifier Telegram

During Phase 2, the Slave sends an Amplifier Telegram (AT) only when it receives a Master Data Telegram (MDT) that contains that Slave's address. The AT contains one Data Record where the length of the Data Block is 0, i.e., the data block is empty. During Phase 2, the Master sends data to each Slave via the Service Channel.

During Phases 3 and 4, each Slave sends an Amplifier Telegram (AT) every SERCOS cycle, at the time designated by the Master during Phase 2. The AT Message Field contains one Data Record, where the length of the Data Block (inside that Data Record) is determined by the data fields configured by the Master during Phase 2. The data fields within the Data Block usually contain feedback and status information.

During Phases 3 and 4, data is sent to the Master by using the Data Blocks (cyclic) or the Service Channel (non-cyclic).



Field	Size (bytes)	Description
BOF	1	Beginning of Frame. The BOF marks the start of a telegram.
ADR	1	sends address.
Status	2	Control word for drive n. Contains operational data.
Service Info	2	Contains the non-cyclic data for drive n.
Operation Data	Variable *	Contains the cyclic data for drive n.
FCS	2	Frame check sequence. Contains circular redundancy check (CRC) info.
EOF	1	End of Frame, which is always 0111 1110. The EOF marks the end of the telegram.

* The length of the Operation Data field is determined during Phase 2.

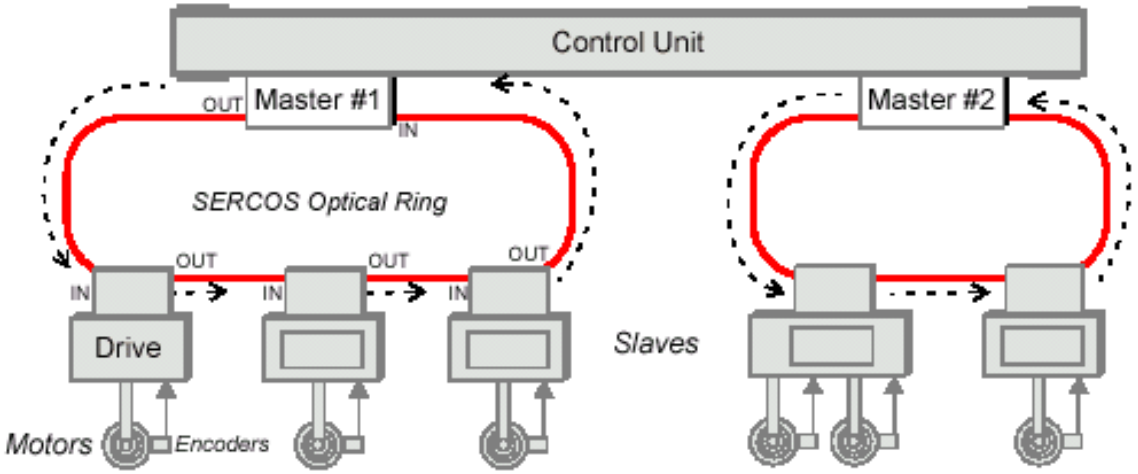
Bit	Name & Value	More Detail
15, 14	Ready to operate 00 drive not ready for power-up 01 drive ready for power-up 10 drive power ready 11 drive ready to operate	
13	Drive Shutdown Error, Class 3 Diags	
12	Change Bit for Class 2 Diags	
11	Change Bit for Class 3 Diags	
9,8	Actual Operation Mode 00 primary op mode 01 secondary op mode 1 10 secondary op mode 2 11 secondary op mode 3	 Defined by S-0-32 Defined by S-0-33 Defined by S-0-34 Defined by S-0-35
7	Real-time Status Bit 2	See S-0-306
6	Real-time Status Bit 1	See S-0-304

TOP

5	Change Bit Commands 0 No Change in Command Status 1 Changing Command Status	
4, 3	**Reserved**	
2	Error 0 No Error 1 Error in Service Channel	Error message is in drive's Service Channel.
1	Busy 0 Step Finished 1 Step in Progress	
0	Master Service Transport Handshake	A Toggle bit

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

Sercos- Application Topologies



SERCOS

A Sercos object represents a Master on the SERCOS ring.

NODE

A Node object represents a Slave Node on the SERCOS ring, which is typically either a drive or an I/O module.

S-X-XXXX (Standard IDNs)

OR

P-X-XXXX (Manufacturer-specific IDNs)

IDN

An Idn object is simply a container for a SERCOS IDN.

IDNLIST — **IDN 1, IDN 2, IDN 3, IDN 4**

An IdnList object keeps a list of Idns.

SERCOS — **IDN 1, IDN 2, IDN 3, IDN 4**

A Sercos object keeps a list of Node objects.

NODE 3 — **IDNLIST** — **IDN 1, IDN 2, IDN 3, IDN 4**

An IdnList object is always associated with a Node.

Introduction | Overview | Data Types | Communications | Procedures | Telegrams | Topologies

Return to Software's Main Menu

SERCOS Communication Error Notification

If your motion controller uses SERCOS, you can take advantage of specialized status information. The MPIStatusMask in the MPIStatus structure makes it possible to read the MPIStatusFlag(s):

```
/* MPIStatus */
typedef enum {
        MPIStatusFlagINVALID,
        MPIStatusFlagCOMM_ERROR,
} MPIStatusFlag;

#define mpiStatusMaskBIT(flag)    (0x1 << (flag))

typedef enum {
    MPIStatusMaskNONE         = 0x0,
    MPIStatusMaskCOMM_ERROR   = mpiStatusMaskBIT(MPIStatusFlagCOMM_ERROR),  /* 0x00000001 */
    MPIStatusMaskMOTOR        = MPIStatusMaskCOMM_ERROR,                     /* 0x00000001 */
    MPIStatusMaskALL          = mpiStatusMaskBIT(MPIStatusFlagLAST) - 1      /* 0x00000001 */
} MPIStatusMask;

typedef struct MPIStatus {
    MPIState        state;
    MPIAction       action;
    MPIEventMask    eventMask;

    long     settled;
    long     atTarget;

    MPIStatusMask     statusMask;
} MPIStatus;
```

Presently, the only MPIStatusFlag supported is MPIStatusFlagCOMM_ERROR. This flag represents the communication status. For XMP-Series SERCOS controllers, this flag represents the status of the SERCOS communication. When the SERCOS communication ring fails, the MPIStatusFlagCOMM_ERROR is set. The flag can only be cleared by re-initializating the SERCOS ring with mpiSercosInit(...).

Return to Sercos Objects page