



# ARINC 825 Library

## Software Manual

## NOTE

The information in this document has been carefully checked and is believed to be entirely reliable. **esd** makes no warranty of any kind with regard to the material in this document, and assumes no responsibility for any errors that may appear in this document. In particular descriptions and technical data specified in this document may not be constituted to be guaranteed product features in any legal sense.

**esd** reserves the right to make changes without notice to this, or any of its products, to improve reliability, performance or design.

All rights to this documentation are reserved by **esd**. Distribution to third parties, and reproduction of this document in any form, whole or in part, are subject to **esd**'s written approval.

© 2015 esd electronic system design gmbh, Hannover

esd electronic system design gmbh  
Vahrenwalder Str. 207  
30165 Hannover  
Germany

Phone: +49-511-372 98-0  
Fax: +49-511-372 98-68  
E-Mail: [info@esd.eu](mailto:info@esd.eu)  
Internet: [www.esd.eu](http://www.esd.eu)

### Trademark Notices

CANopen<sup>®</sup> and CiA<sup>®</sup> are registered community trademarks of CAN in Automation e.V.  
Windows<sup>®</sup> is a registered trademark of Microsoft Corporation in the United States and other countries.  
Linux<sup>®</sup> is the registered trademark of Linus Torvalds in the United States and/or other countries.  
QNX<sup>®</sup> is a registered trademark of QNX Software Systems Limited, and is registered trademark and/or used in certain jurisdictions.

All other trademarks, product names, company names or company logos used in this manual are reserved by their respective owners.

<b>Document file:</b>	I:\Texte\Doku\MANUALS\PROGRAM\CAN\C.1140.21_ARINC825\ARINC825_Library_Manual_en_14.odt
<b>Date of print:</b>	2015-11-23

**Documentation valid for software versions:**

ARINC 825 Library for operating system...	Order no. *	Documented Software Rev.
Windows® 8, Windows 7, Windows XP, Windows 2000	C.1140.06 (Windows/Linux CD)	ARINC Library: 1.1.23
Linux®		
RTX/RTX64®	C.1140.16	
QNX®	C.1140.17	
VxWorks®	C.1140.18	ARINC Library: 1.1.24

\* See order information, page 105.

**Document History**

The changes in the document listed below affect changes in the hardware as well as changes in the description of the facts, only.

Revision	Chapter	Changes versus previous version	Date
1.2	-	Trademark Notices inserted	2014-02-11
	1.	Overview updated	
	2.	Chapter about NTCAN added	
	3. - 7.	Chapters moved and updated according to ARINC 825 Library Rev. 1.1.09	
	8.	New chapter contains ARINC LABVIEW Library description	
	9.	New chapter: "Order Information"	
1.3	-	RTX/RTX64 support	2014-09-29
	7.12 - 7.15	New chapters describe <b>arincRxStart</b> , <b>arincRxStop</b> , <b>arincTxStart</b> and <b>arincTxStop</b> functions as alternative for <b>arincScheduleStart</b> and <b>arincScheduleStop</b>	
	9.	Order Information revised	
1.4	8.5.11	New chapter: ArincRxStart	2015-11-23
	8.5.12	New chapter: ArincRxStop	
	8.5.20	New chapter: ArincTxStart	
	8.5.21	New chapter: ArincTxStop	

Technical details are subject to change without further notice.

---

## Table of contents

1. Overview .....	6
2. NTCAN .....	7
3. ARINC-Time .....	8
4. CAN-errors .....	9
5. Defines .....	12
5.1 CAN-IDs .....	13
5.2 CAN-Length .....	14
5.3 CAN-Baudrate .....	15
5.4 Status-Busstates .....	16
5.5 CAN-Controller .....	17
5.6 Errorcodes .....	18
5.7 Errorformats .....	19
5.8 Thread-Priority .....	20
6. Datatypes .....	21
6.1 ARINC_CMSG_T .....	22
6.2 ARINC_STATUS .....	24
6.3 ARINC_ERROR .....	26
6.4 ARINC_RESULT .....	28
6.5 ARINC_HANDLE .....	29
6.6 ARINC_GROUP .....	30
6.7 ARINC_ERROR_HANDLER .....	31
7. Functions .....	32
7.1 arincHandleOpen .....	33
7.2 arincClose .....	34
7.3 arincBaudrateSet .....	35
7.4 arincBaudrateGet .....	36
7.5 arincStatus .....	37
7.6 arincStatusReset .....	38
7.7 arincFormatError .....	39
7.8 arincErrorHandler .....	40
7.9 arincTimeGet .....	41
7.10 arincIntervalSet .....	42
7.11 arincIntervalGet .....	43
7.12 arincRxStart .....	44
7.13 arincRxStop .....	45
7.14 arincTxStart .....	46
7.15 arincTxStop .....	48
7.16 arincScheduleStart .....	49
7.17 arincScheduleStop .....	51
7.18 arincPoll .....	52
7.19 arincTxObjUpdate .....	53
7.20 arincTxObjDisable .....	54
7.21 arincWaitForTimeslot .....	55
7.22 arincObjAdd .....	56
7.23 arincObjDelete .....	57
8. ARINC825 LabVIEW Library.....	58
8.1 Archive contents.....	59
8.2 Installation.....	60
8.3 Basic Usage Information.....	61

8.4	A825 VIs (Signal Based VIs)	62
8.4.1	A825Error2String	63
8.4.2	A825Info	64
8.4.3	A825ObjectPoll	65
8.4.4	A825ObjectSend	66
8.4.5	A825ObjectTrigger	67
8.4.6	A825ProjectClose	68
8.4.7	A825ProjectOpen	69
8.4.7.1	Example of a Project File	70
8.4.8	A825ScheduleStart	72
8.4.9	A825ScheduleStop	73
8.4.10	A825SignalPoll	74
8.4.11	A825SignalTrigger	75
8.4.12	A825SignalUpdate	76
8.4.13	A825Status	77
8.4.14	A825StatusReset	78
8.4.15	A825TimeGet	79
8.4.16	A825TxObjectDisable	80
8.4.17	A825TxObjectUpdate	81
8.5	ARINC VIs (Native VIs)	82
8.5.1	ArincBaudrateGet	83
8.5.2	ArincBaudrateSet	84
8.5.3	ArincClose	85
8.5.4	ArincError2String	86
8.5.5	ArincHandleOpen	87
8.5.6	ArincIntervalGet	88
8.5.7	ArincIntervalSet	89
8.5.8	ArincObjAdd	90
8.5.9	ArincObjDelete	91
8.5.10	ArincPoll	92
8.5.11	ArincRxStart	93
8.5.12	ArincRxStop	94
8.5.13	ArincScheduleStart	95
8.5.14	ArincScheduleStop	96
8.5.15	ArincStatus	97
8.5.16	ArincStatusReset	98
8.5.17	ArincTimeGet	99
8.5.18	ArincTxObjDisable	100
8.5.19	ArincTxObjUpdate	101
8.5.20	ArincTxStart	102
8.5.21	ArincTxStop	103
8.5.22	ArincWaitForTimeslot	104
9.	Order Information	105

# 1. Overview

[ [Top](#) ] [ [Generics](#) ]

## NAME

ARINC825 Library

## FILE NAME

arinc825.h

## BRIEF MODULE DESCRIPTION

Header for ARINC825 support library

## AUTHOR

Andreas Block (BL)

## CREATION DATE

25-May-2009

## PORTABILITY

ANSI-C

vxWorks, Linux, Win32, RTX/RTX64

Depends on [NTCAN](#)- and PSYS-library

## COPYRIGHT

Copyright (c) 2009-2014 by esd electronic system design gmbh

This software is copyrighted by and is the sole property of esd gmbh. All rights, title, ownership, or other interests in the software remain the property of esd gmbh. This software may only be used in accordance with the corresponding license agreement. Any unauthorized use, duplication, transmission, distribution, or disclosure of this software is expressly forbidden.

This Copyright notice may not be removed or modified without prior written consent of esd gmbh.

esd gmbh, reserves the right to modify this software without notice.

electronic system design gmbh  
Vahrenwalder Str 207  
30165 Hannover  
Germany

Tel. +49-511-37298-0  
Fax. +49-511-37298-68  
<http://www.esd.eu>  
sales@esd.eu

---

## 2. NTCAN

[ [Top](#) ] [ [Generics](#) ]

### DESCRIPTION

esd provides a general API to program CAN interfaces, called **NTCAN**. ARINC825 library makes heavy use of **NTCAN** and could be seen as an extension on top of **NTCAN**. Thus you'll see lots of references to **NTCAN** in this documentation, as for example standard **NTCAN** error codes might be returned by ARINC-825 functions. Where feasible (and noted within this documentation) you can also use the respective **NTCAN** defines as parameters.

**NTCAN** documentation can be found on esd's website ([www.esd.eu](http://www.esd.eu)) or can be requested directly from esd's support team ([support@esd.eu](mailto:support@esd.eu)).

## 3. ARINC-Time

[ [Top](#) ] [ [Generics](#) ]

### DESCRIPTION

In general time is handled as "time ticks". Depending on your CAN hardware the frequency of these ticks differs. Once you open a ARINC825 handle via [arincHandleOpen\(\)](#) you can retrieve the tick frequency in use for this CAN bus. In this way it is very well possible to have different CAN boards with differing time tick frequencies in your system.

The timestamp is monotonically increasing (please see notes below). Times can be added or subtracted without problems.

### NOTES

esd also delivers special CAN hardware, which provides connection to e.g. IRIG-B. Whenever such hardware is used, you need to take care, that your timebase is stable, before ARINC825 scheduler is started. Please see documentation accompanied with such hardware on how to accomplish this.

### SEE ALSO

[arincHandleOpen\(\)](#)  
[arincTimeGet\(\)](#)



## 4. CAN-errors

[ [Top](#) ] [ [Generics](#) ]

### DESCRIPTION

Basics:

-----

On CAN bus there are sophisticated error handling techniques, which not only allow to detect errors, but also help to remove defective hardware from a CAN bus in order to keep the remaining system working. This is called CAN error confinement.

It is accomplished via two error counters (RX + TX, called RX Error Counter (REC) and TX Error Counter (TEC)) and a ruleset related to these, defining when these counters need to be in- or decremented. This won't be discussed in all details here, as there's lots of literature written and web resources published on this topic.

Depending on these error counters every CAN node is in one of four states:

BUS OK (REC < 96 and TEC < 96)

Normal state of a CAN node, when everything is ok. This does NOT necessarily mean the absence of errors on CAN bus, though. The node is also said to be "error active", as it will actively propagate (or globalize) any detected error.

BUS WARN (96 >= REC < 128 or 96 >= TEC < 128)

At least one error counter has reached "error warning limit", there's a significant amount of errors on CAN bus, yet overall function doesn't seem to be affected. The node remains "error active" in this state.

Note: This state is not described in ISO11898, yet most CAN controllers support it.

ERROR PASSIVE (128 >= REC <= 255 or 128 >= TEC <= 255)

Contrary to the first two states a CAN node goes "error passive" (opposed to "error active") by exceeding the next threshold with any of its error counters. The node is still able to transmit and receive CAN frames, but it will not any longer propagate detected errors over the CAN bus.

BUS OFF (TEC > 255)

Lastly a node enters bus off state, where it will no longer take part in CAN communication.

In general the error counters will be incremented, whenever an error is detected on CAN bus. Most CAN controllers provide special means to further investigate the circumstances of such errors. Amongst other information the type of error and position of the error condition within a CAN frame are stored in an error code capture register (ECC).

Means of error detection:

-----

ARINC825 library provides several different means to detect CAN errors or to be notified about them.

(A) Polling:

At any given time the CAN status can be polled using [arincStatus\(\)](#) or with [NTCAN-API](#) call `canIoctl(NTCAN_IOCTL_GET_CTRL_STATUS)`. The [ARINC\\_STATUS](#) structure returned from [arincStatus\(\)](#) also contains ECC information of the last CAN bus error (see below).

(B) Asynchronous notification with error handler:

By means of [arincErrorHandler\(\)](#) an asynchronous error handler function can be registered. On occurrence of certain errors (amongst others CAN bus errors)

## CAN-errors

the registered handler will be called and is also passed an `ARINC_STATUS` structure with ECC information.

(C) Asynchronous notification by virtual CAN events:

There're two CAN events related to errors on CAN bus (see `NTCAN` documentation). These events are received as virtual CAN frames via `canRead()/canTake()`, if the respective CAN IDs have been enabled with `canIdAdd()`. The first one is the CAN error event, which is generated on every state change in a node's CAN state machine. The second one, called extended error event, is generated on every CAN error frame on CAN bus. There's no guarantee to receive one event per error frame, as the driver analyzes the ECC information and tries to prevent IRQ floods by disabling the interrupt for certain time periods if heavy load is detected. Nevertheless in real world scenarios chances are quite good, you'll receive an event for every CAN error frame.

There are data structures defined in `NTCAN` to ease evaluation of these events (`EV_CAN_ERROR` and `EV_CAN_ERROR_EXT`), these can be simply mapped to the data section of the respective CAN event frames.

Additionally `canFormatEvent()` (again from `NTCAN`) can be used to convert the contained information into human readable strings.

### ECC Byte

-----

In order to further analyze a CAN bus error the ECC (error code capture) is of special importance. This information is heavily hardware dependent and you may need the documentation of the involved CAN controller to make full use of this feature.

The two most common CAN controllers in esd products are esd's esdACC and NXP's SJA1000. Both controllers share the same bit encoding, so decoding of the ECC byte works exactly the same and shall be described here in a bit more detail:

Bit	Symbol	Name	Value	Function
ECC Bit 7 (MSB)	ERRC1	Error Code 1	-	-
ECC Bit 6	ERRC0	Error Code 2	-	-
ECC Bit 5	DIR	Direction	1	RX; error occurred during reception
			0	TX; error occurred during transmission
ECC Bit 4	SEG4	Segment 4	-	-
ECC Bit 3	SEG3	Segment 3	-	-
ECC Bit 2	SEG2	Segment 2	-	-
ECC Bit 1	SEG1	Segment 1	-	-
ECC Bit 0 (LSB)	SEG0	Segment 0	-	-

### Error type

-----

ERRC1	ERRC0	Function
0	0	bit error
0	1	form error
1	0	stuff error
1	1	other type of error

### Error position

SEG4	SEG3	SEG2	SEG1	SEG0	Function
0	0	0	1	1	start of frame
0	0	0	1	0	ID.28 to ID.21
0	0	1	1	0	ID.20 to ID.18
0	0	1	0	0	bit SRTR
0	0	1	0	1	bit IDE
0	0	1	1	1	ID.17 to ID.13
0	1	1	1	1	ID.12 to ID.5
0	1	1	1	0	ID.4 to ID.0
0	1	1	0	0	bit RTR
0	1	1	0	1	reserved bit 1
0	1	0	0	1	reserved bit 0
0	1	0	1	1	data length code
0	1	0	1	0	data field
0	1	0	0	0	CRC sequence
1	1	0	0	0	CRC delimiter
1	1	0	0	1	acknowledge slot
1	1	0	1	1	acknowledge delimiter
1	1	0	1	0	end of frame
1	0	0	1	0	intermission
1	0	0	0	1	active error flag
1	0	1	1	0	passive error flag
1	0	0	1	1	tolerate dominant bits
1	0	1	1	1	error delimiter
1	1	1	0	0	overload flag

## SEE ALSO

[ARINC\\_STATUS](#)  
[arincStatus\(\)](#)  
[arincErrorHandler\(\)](#)

## 5. Defines

[ [Top](#) ] [ [Generics](#) ]

### DESCRIPTION

Documentation of symbol definitions:

- [CAN-IDs](#)
- [CAN-Length](#)
- [CAN-Baudrate](#)
- [Status-Busstates](#)
- [CAN-Controller](#)
- [Errorcodes](#)
- [Errorformats](#)
- [Thread-Priority](#)

## 5.1 CAN-IDs

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

### DESCRIPTION

Whenever **CAN-IDs** are involved and are used as input parameters and/or return values, all CAN-ID defines from [NTCAN](#) can be used.

### NOTES

Although ARINC825 is specified only for 29-Bit **CAN-IDs**, it is possible to set up ARINC825 schedules with objects with 11-Bit CAN-ID. This is an option for your convenience, but shouldn't be used on strict ARINC825 busses (it wouldn't make much sense there anyway...).

### SEE ALSO

[ARINC\\_CMSG\\_T](#)

### 5.2 CAN-Length

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

#### DESCRIPTION

The CAN Data Length Code (DLC) does not only contain the number of data bytes contained in a CAN frame, but there might be additional information encoded into this field (e.g. RTR frames).

All defines from [NTCAN-API](#) apply here and may be used as input parameters as well as to evaluate return values.

ARINC825 library extends these defines by one more: `ARINC_OLD_DATA`

```
ARINC_OLD_DATA    -- Marks old data
                   When using arincPoll\(\) this flag can be used to quickly
                   identify data objects, which haven't changed since last
                   call of arincPoll\(\).
```

#### SEE ALSO

[ARINC\\_CMSG\\_T](#)

## 5.3 CAN-Baudrate

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

### DESCRIPTION

In order to communicate on CAN bus a CAN node needs to have a baudrate set. This can be accomplished via [arincBaudrateSet\(\)](#) and [arincBaudrateGet\(\)](#). [NTCAN](#) API contains a whole bunch of defines to set predefined baudrates recommended by CiA, as well as defines to set baudrate numerically (in bits per second) or program the bit timing registers of the CAN controller directly.

### NOTES

Please beware, configuring a wrong baudrate will severely affect the CAN communication. It is advised to check, if another application or another handle in the same application has already set a baudrate (via [arincBaudrateGet\(\)](#)). If absolutely unsure you can make use of auto-baud feature of esd CAN drivers. See [NTCAN](#) documentation for further information on baudrates and auto-baud.

### SEE ALSO

[arincBaudrateSet\(\)](#)  
[arincBaudrateGet\(\)](#)

### 5.4 Status-Busstates

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

#### DESCRIPTION

[Defines](#) to decode CAN bus status as delivered by [arincStatus\(\)](#), [arincErrorHandler\(\)](#) or `NTCAN_EV_CAN_ERROR` and `canIoctl(NTCAN_IOCTL_GET_CTRL_STATUS)` can be found in [NTCAN](#) header.

#### SEE ALSO

[CAN-errors](#)  
[arincErrorHandler\(\)](#)  
[arincStatus\(\)](#)



## 5.5 CAN-Controller

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

### DESCRIPTION

The type of CAN controller is returned from `arincStatus()`. `NTCAN` provides more means to determine the controller type (`canIoctl(NTCAN_IOCTL_GET_CTRL_STATUS)` and `canIoctl(NTCAN_IOCTL_GET_BITRATE_DETAILS)`) as well as defines to decode the controller type.

### SEE ALSO

`arincStatus()`  
`ARINC_STATUS`

## 5.6 Errorcodes

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

### NAME

Error codes generated by ARINC825 library functions

### ERRORS

```

ARINC_ERRNO_BASE          -- Whenever possible, ARINC825 library uses
                           system error codes (e.g. under Linux the
                           errno values will be used) or NTCAN error
                           codes. For errors, which can't be mapped
                           to these system codes, ARINC_ERRNO_BASE is
                           used as offset to avoid ambiguities.

ARINC_SUCCESS             -- No error

ARINC_ERROR_TXSLICE_TIMEOUT -- The next TX slice was not triggered.
ARINC_ERROR_TXSLICE_INCOMPLETE -- A TX slice wasn't transmitted completely.
ARINC_ERROR_SCHED_DISABLED -- The scheduling hasn't been enabled, yet.
                           Use arincScheduleStart\(\) or arincTxStart\(\)
first.
ARINC_ERROR_SCHED_ENABLED -- The scheduling is currently enabled.
                           Use arincScheduleStop\(\) or arincTxStop\(\) first.
ARINC_ERROR_ID_BUSY      -- There's already an object added with the
                           same CAN ID.
ARINC_ERROR_COL_BUSY     -- There's already an object with the same
                           group-m-n combination.
ARINC_ERROR_ID_NOT_FOUND -- A referenced ARINC object was not found
                           (e.g. with arincTxObjUpdate\(\)).
ARINC_ERROR_TIME_NOT_SET -- Failure while configuring start time and/or
                           time slice duration for scheduling.
ARINC_ERROR_NO_INTERVAL  -- Attempt to start scheduling without a valid
                           timeslice duration.
ARINC_ERROR_NOT_TX       -- Attempt to update data of a RX object.
ARINC_CAN_STATE_CHANGE   -- CAN bus state transition has occurred
                           (which is not necessarily an error condition).
ARINC_CAN_ERROR          -- A CAN error frame has occurred on bus.
ARINC_ERROR_INTERVAL_LOW -- Attempt to configure an timeslice duration
                           equal or below 1ms.

```

### NOTES

These error codes are unique to ARINC825 functions, nevertheless as documented with each function all [NTCAN](#) error codes may occur as well.

Whenever ARINC library makes use of functions provided by the underlying operating system, the respective system error codes are passed to the user. Since it is not always well documented, which error codes might be thrown in these cases, these codes might not always be documented below.

You can use [arincFormatError\(\)](#) to get a string representation for the described errors.

#### SSE ALSO

[arincFormatError\(\)](#)

## 5.7 Errorformats

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

### DESCRIPTION

`NTCAN` defines special values to select the output type of `arincFormatError()`.

### SEE ALSO

`arincFormatError()`

### 5.8 Thread-Priority

[ [Top](#) ] [ [Defines](#) ] [ [Definitions](#) ]

#### NAME

Special values to be used with the priority parameters of [arincScheduleStart\(\)](#)

#### DESCRIPTION

<code>ARINC_PRIO_HIGH</code>	-- Use a high priority for maximum accuracy of scheduling
<code>ARINC_PRIO_INHERIT</code>	-- Inherit the priority of the calling application
<code>ARINC_PRIO_SET</code>	-- Set system dependent thread priorities

#### SEE ALSO

[arincScheduleStart\(\)](#)  
[arincRxStart\(\)](#)  
[arincTxStart\(\)](#)

## 6. Datatypes

[ [Top](#) ] [ [Generics](#) ]

### DESCRIPTION

Definitions of data types:

- [ARINC\\_CMSG\\_T](#)
- [ARINC\\_STATUS](#)
- [ARINC\\_ERROR](#)
- [ARINC\\_RESULT](#)
- [ARINC\\_HANDLE](#)
- [ARINC\\_GROUP](#)
- [ARINC\\_ERROR\\_HANDLER](#)

### 6.1 ARINC\_CMSG\_T

[ [Top](#) ] [ [Datatypes](#) ] [ [Structures](#) ]

#### NAME

**ARINC\_CMSG\_T** -- CAN object structure for use with ARINC825 functions

#### DESCRIPTION

Stores ARINC825 and CAN frame data

#### SYNOPSIS

```
struct _ARINC_CMSG_T {
    INT32      id;
    UINT8      len;
    UINT8      msg_lost;
    UINT8      reserved[2];
    UINT8      data[8];
    UINT64     timestamp;
    ARINC\_GROUP group;
    INT32      m;
    INT32      n;
    INT32      countTx;
    INT32      countRx;
}
```

#### ATTRIBUTES

id -- CAN ID, defines priority on CAN bus, can be combined with CAN ID defines (see [CAN-IDs](#) and [NTCAN](#) docs)

len -- Number of data bytes contained within CAN frame (0..8) (see [CAN-Length](#) and [NTCAN](#) docs for additional defines to be used with this field)

msg\_lost -- On reception the number of lost RX frames is returned here. With modern CAN hardware this should be zero. Nevertheless it's recommended to evaluate this field, if you rely on streams of data for example.

data -- Up to eight data bytes

timestamp -- 64-Bit timestamp (frequency is returned from [arincHandleOpen\(\)](#))

Additional ARINC information:

group -- ARINC825 group, see [ARINC\\_GROUP](#)

m -- ARINC825 column, it's the m'th object within it's group (this needs to be unique for a given "group-n" combination)

n -- ARINC825 slice index, for groups > 0 the object is located in the n'th timeslice

countTx -- Number of times this object was transmitted  
Positive values: The last n times the object was transmitted successfully  
Negative values: n failures to send this object

countRx -- Number of times this object was received

#### NOTES

It is advised to evaluate countTx on ARINC825 TX objects and countRx on RX objects. Although both counters are kept up to date for both types of objects and a comparison might be used to reveal certain error conditions, one needs to take a possible deviance of one frame into account. This is caused by the fact, that transmission and reception of the transmitted frame do not happen simultaneously, reception always follows transmission. Although it would be technically possible to synchronize both events and atomically increment both counters at the same time, the performance impact

of a rather long locked path would be drastic. Another workaround would be incrementing the RX counter on the event of successful transmission. Since this would forego the chance of an extra step of verification, the deviation of both counters is deliberately accepted. Nevertheless if everything works correctly both counters shouldn't differ by any more than one frame.

**SEE ALSO**

[arincPoll\(\)](#)  
[arincTxObjUpdate\(\)](#)  
[arincObjAdd\(\)](#)  
[arincObjDelete\(\)](#)

## 6.2 ARINC\_STATUS

[ [Top](#) ] [ [Datatypes](#) ] [ [Structures](#) ]

### NAME

**ARINC\_STATUS** -- Status structure for use with [arincStatus\(\)](#)

### DESCRIPTION

Stores version-, status- and error information.

### SYNOPSIS

```
struct _ARINC_STATUS {
    UINT16    hardware;
    UINT16    firmware;
    UINT16    driver;
    UINT16    dll;
    UINT32    boardstatus;
    UINT8     boardid[14];
    UINT16    features;
    UINT16    dllarinc;
    UINT16    reserved;
    UINT64    time;
    UINT32    rxCount;
    UINT64    rxLastTime;
    UINT32    txCount;
    UINT64    txLastTime;
    UINT32    errorCount;
    INT32     errorLast;
    UINT64    errorLastTime;
    UINT32    errorCode;
    UINT64    errorCodeTime;
    UINT32    reserved2;
    UINT8     canStatus;
    UINT8     canErrorCountRx;
    UINT8     canErrorCountTx;
    UINT8     reserved3;
    UINT32    errorLostFrames;
}
```

### ATTRIBUTES

hardware	-- Hardware version of CAN hardware
firmware	-- Firmware version of CAN hardware (if applicable, zero otherwise)
driver	-- CAN driver version
dll	-- <a href="#">NTCAN</a> API library version
boardstatus	-- Overall status of CAN hardware Most significant byte contains type of CAN controller (see <a href="#">CAN-Controller</a> )
boardid	-- 14-Byte long string, containing the name of the CAN hardware
features	-- 16-Bit wide flag field, specifying features supported by CAN hardware (see defines in <a href="#">NTCAN</a> to decode these)
dllarinc	-- Version of this ARINC825 library
reserved	-- Reserved for future use, aligns structure
time	-- Current time
rxCount	-- (*) Number of ARINC825 objects received on a certain handle/CAN bus (including those sent by the node itself)
rxLastTime	-- (*) Time of last reception
txCount	-- (*) Number of ARINC825 objects transmitted on a certain



---

```

                                handle/CAN bus
txLastTime      -- (*) Time of last transmission
errorCount      -- (*) Overall number of errors on a certain handle/CAN bus
errorLast       -- (*) The last error, which occurred (see Errorcodes)
                (Note: These are not necessarily CAN bus errors)
errorLastTime   -- (*) Approximated time of last error
errorCode       -- (*) Detailed info on the last CAN bus error
errorCodeTime   -- (*) Time of last CAN bus error
canStatus       -- Status of CAN bus (see Status-Busstates)
canErrorCountRx -- CAN RX error counter (see "Rules of error confinement"
                in chapter 13.1.4 of ISO 11898-1)
canErrorCountTx -- CAN TX error counter (see "Rules of error confinement"
                in chapter 13.1.4 of ISO 11898-1)
errorLostFrames -- (*) Frames lost by either CAN controller or CAN driver

```

## NOTES

(\*) - Special ARINC825 status-/error information. These fields can be reset to zero using [arincStatusReset\(\)](#)

Version information (hardware, firmware, driver, dll, dllarinc) is returned in the following format:

0xXYZZ with 0xX = major version, 0xY = minor version and 0xZZ = change level

A format string to print version info for example could look like this:

```
("%.%d.%.%d", (v >> 12), ((v & 0x0F00) >> 8), (v & 0x00FF))
```

## SEE ALSO

[CAN-errors](#)

[arincStatus\(\)](#)

[arincStatusReset\(\)](#)

## 6.3 ARINC\_ERROR

[ [Top](#) ] [ [Datatypes](#) ] [ [Structures](#) ]

### NAME

**ARINC\_ERROR** -- Error structure passed as an argument to an error handler

### DESCRIPTION

If the user connects an error handler function, this will be called asynchronously on occurrence of an error. The argument of this handler points to an **ARINC\_ERROR** structure, which contains more details on the type and the circumstances of the error.

### SYNOPSIS

```
struct _ARINC_ERROR {
    void *userParam;
    UINT32  rxCount;
    UINT64  rxLastTime;
    UINT32  txCount;
    UINT64  txLastTime;
    UINT32  errorCount;
    INT32   errorLast;
    UINT64  errorLastTime;
    UINT32  errorCode;
    UINT64  errorCodeTime;
    UINT32  reserved;
    UINT8   canStatus;
    UINT8   canErrorCountRx;
    UINT8   canErrorCountTx;
    UINT8   reserved2;
    UINT32  errorLostFrames;
}
```

### ATTRIBUTES

userParam	-- When registering an error handler, the user may specify a pointer, which is passed to the error handler without modification. This provides the user with the means to pass any kind of data to the error handler (e.g. some way to synchronize the error handler with the rest of an application).
rxCount	-- Number of ARINC825 objects received on a certain handle/CAN bus (including the ones send by this node itself)
rxLastTime	-- Time of last reception
txCount	-- Number of ARINC825 objects transmitted on a certain handle/CAN bus
txLastTime	-- Time of last transmission
errorCount	-- Overall number of errors on a certain handle/CAN bus
errorLast	-- The last error, which occurred (see <a href="#">Errorcodes</a> ) (Note: These are not necessarily CAN bus errors)
errorLastTime	-- Approximated time of last error
errorCode	-- Detailed info on the last CAN bus error
errorCodeTime	-- Time of last CAN bus error
canStatus	-- Status of CAN bus (see <a href="#">Status-Busstates</a> )
canErrorCountRx	-- CAN RX error counter (see "Rules of error confinement" in chapter 13.1.4 of ISO 11898-1)
canErrorCountTx	-- CAN TX error counter (see "Rules of error confinement" in chapter 13.1.4 of ISO 11898-1)
reserved	-- Reserved for future use, aligns structure

`errorLostFrames` -- Frames lost by either CAN controller or CAN driver

**SEE ALSO**

[ARINC\\_ERROR\\_HANDLER](#)  
[arincErrorHandler\(\)](#)

### 6.4 ARINC\_RESULT

[ [Top](#) ] [ [Datatypes](#) ] [ [Types](#) ]

#### NAME

**ARINC\_RESULT** -- Return type of all functions within this library

#### DESCRIPTION

On success functions within this library return ARINC\_SUCCESS, otherwise one of the error codes above ([Errorcodes](#) and [NTCAN](#) error codes).

## 6.5 ARINC\_HANDLE

[ [Top](#) ] [ [Datatypes](#) ] [ [Types](#) ]

### NAME

**ARINC\_HANDLE** -- Handle to address a certain CAN bus

### DESCRIPTION

A handle needs to be created using [arincHandleOpen\(\)](#). Every function within this library needs to be passed a valid handle. By this means the functions are connected to a certain physical CAN bus.

### NOTES

ARINC\_HANDLES can be imagined as virtual CAN nodes. It is possible to open multiple handles and each handle might have its own parameter set. A handle can be shared by multiple threads, in such a case some restrictions need to be observed. Foremost only one thread should configure and start an ARINC825 scheduling. Further restrictions will be noted together with the function descriptions below.

If multiple handles are used on the same CAN node, care must be taken, that only one handle configures the ARINC825 scheduling with [arincIntervalSet\(\)](#) and starts it with [arincScheduleStart\(\)](#).

It is NOT possible to have multiple differently configured ARINC825 schedulers on a single CAN node. The results will be undefined.

When a handle is no longer needed, it should be disposed using [arincClose\(\)](#), in order to free system resources allocated by the handle.

### SEE ALSO

[arincHandleOpen\(\)](#)  
[arincClose\(\)](#)

### 6.6 ARINC\_GROUP

[ [Top](#) ] [ [Datatypes](#) ] [ [Types](#) ]

#### NAME

**ARINC\_GROUP** -- Type of ARINC groups

#### DESCRIPTION

Use for member "group" of [ARINC\\_CMSG\\_T](#) struct.

Generally speaking, an ARINC group specifies the rate by which ARINC objects get repeated in (group+1) multiples of timeslice intervals.

For example:

Setting group to zero leads to objects within this group being transmitted every timeslice.

Setting group to four leads to objects within this group being transmitted every fifth timeslice.

There's one special group value: `ARINC_GROUP_RX`

Using this as group value, defines the object as "receive only".

#### SEE ALSO

[ARINC\\_CMSG\\_T](#)  
[arincTxObjUpdate\(\)](#)  
[arincPoll\(\)](#)  
[arincObjAdd\(\)](#)  
[arincObjDelete\(\)](#)

## 6.7 ARINC\_ERROR\_HANDLER

[ [Top](#) ] [ [Datatypes](#) ] [ [Types](#) ]

### NAME

**ARINC\_ERROR\_HANDLER** -- Function pointer type for asynchronous error handler

### DESCRIPTION

A function of this type can be registered using [arincErrorHandler\(\)](#), in order to be asynchronously notified of any errors.

### SEE ALSO

[ARINC\\_ERROR](#)  
[arincErrorHandler\(\)](#)

## 7. Functions

[ [Top](#) ] [ [Generics](#) ]

### DESCRIPTION

Library function descriptions

- [arincHandleOpen\(\)](#)
- [arincClose\(\)](#)
- [arincBaudrateSet\(\)](#)
- [arincBaudrateGet\(\)](#)
- [arincStatus\(\)](#)
- [arincStatusReset\(\)](#)
- [arincFormatError\(\)](#)
- [arincErrorHandler\(\)](#)
- [arincTimeGet\(\)](#)
- [arincIntervalSet\(\)](#)
- [arincIntervalGet\(\)](#)
- [arincRxStart\(\)](#)
- [arincRxStop\(\)](#)
- [arincTxStart\(\)](#)
- [arincTxStop\(\)](#)
- [arincScheduleStart\(\)](#)
- [arincScheduleStop\(\)](#)
- [arincPoll\(\)](#)
- [arincTxObjUpdate\(\)](#)
- [arincTxObjDisable\(\)](#)
- [arincWaitForTimeslot\(\)](#)
- [arincObjAdd\(\)](#)
- [arincObjDelete\(\)](#)



## 7.1 arincHandleOpen

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincHandleOpen**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincHandleOpen(INT32 net,
                                                    ARINC_HANDLE *pHnd,
                                                    UINT64 *pFreq)
```

### FUNCTION

Opens a handle to access a certain CAN bus.

### PARAMETERS

net -- IN:  
The net number assigned to the desired CAN bus.

pHnd -- OUT:  
Points to an [ARINC\\_HANDLE](#) variable, which will be used to store the newly created handle.

pFreq -- OUT:  
Points to an [UINT64](#) variable, which will be used to store the time tick frequency. This can be set to NULL, if the frequency is not needed.

### RESULT

Success: [ARINC\\_SUCCESS](#)

Error: [NTCAN\\_INVALID\\_PARAMETER](#) - pHnd is NULL or one of the other parameters out of range

[NTCAN\\_INSUFFICIENT\\_RESOURCES](#) - Not enough memory to allocate all resources needed for the new handle

[NTCAN\\_NET\\_NOT\\_FOUND](#) - The specified net wasn't found

### SEE ALSO

[arincClose\(\)](#)  
For some more notes on handles, have a look at [ARINC\\_HANDLE](#).

### 7.2 arincClose

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincClose**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincClose(ARINC_HANDLE hnd)
```

#### FUNCTION

Closes a handle, which was previously opened by [arincHandleOpen\(\)](#). All resources allocated by this handle will be freed, ARINC objects will be deleted.

#### NOTES

A baudrate configured with this handle using [arincBaudrateSet\(\)](#), won't be reset on **arincClose()**. This has to be done explicitly using [arincBaudrateSet\(\)](#).

#### PARAMETERS

hnd -- IN:  
The handle, that needs to be closed.

#### RESULT

Success: ARINC\_SUCCESS  
Error: NTCAN\_INVALID\_HANDLE - An invalid handle was used

#### SEE ALSO

[arincHandleOpen\(\)](#)

## 7.3 arincBaudrateSet

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincBaudrateSet**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincBaudrateSet(ARINC_HANDLE hnd,  
                                                    UINT32          baud)
```

### FUNCTION

Configures the baudrate for the CAN bus belonging to the given handle. Several defines can be used (see [CAN-Baudrate](#)) in order to ease configuration of a certain baudrate.

### NOTES

Baudrate obviously is an attribute of the underlying CAN node and not of the handle itself. In order to have other handles asynchronously notified about the change in baudrate, one can make use of the "baudrate change event" in [NTCAN](#) (please have a look at [NTCAN](#) docs).

### PARAMETERS

hnd -- IN:

A valid handle, which is associated with the CAN bus to be reconfigured.

baud -- IN:

The desired baudrate. It can be set in many ways (e.g. using indices for predefined baudrates, specifying baudrates numerically or programming the BTRs of the CAN controller directly), please see [NTCAN](#) docs.

### RESULT

Success: ARINC\_SUCCESS

Error: NTCAN\_INVALID\_HANDLE - An invalid handle or baudrate was used

### SEE ALSO

[CAN-Baudrate](#)

[arincBaudrateGet\(\)](#)

### 7.4 arincBaudrateGet

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincBaudrateGet**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincBaudrateGet(ARINC_HANDLE hnd,  
                                                    UINT32 *pBaud)
```

#### FUNCTION

Returns the baudrate configured for the CAN bus, which is associated with the given handle.

#### NOTES

The baudrate is returned in the same format as it was formerly set by [arincBaudrateSet\(\)](#). Since baudrate is an attribute of the underlying CAN node and not of the handle itself, this function can be used to detect changes of the baudrate of the CAN bus or to prevent re- and/or misconfiguration.

#### PARAMETERS

hnd -- IN:  
A valid handle, which is associated with the CAN bus in question.  
pBaud -- OUT:  
The baudrate is returned within the variable pointed to by pBaud.

#### RESULT

Success: ARINC\_SUCCESS  
Error: NTCAN\_INVALID\_HANDLE - An invalid handle  
NTCAN\_INVALID\_PARAMETER - An invalid baudrate was used

#### SEE ALSO

[CAN-Baudrate](#)  
[arincBaudrateSet\(\)](#)

## 7.5 arincStatus

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincStatus**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincStatus(ARINC_HANDLE hnd,  
                                               ARINC_STATUS *pStatus)
```

### FUNCTION

Returns information about the current state of the CAN bus, software versions and special ARINC825 information (e.g. last error and time of occurrence).

### PARAMETERS

hnd -- IN:

A valid handle, which is associated with the CAN bus in question.

pStatus -- OUT:

A pointer to a [ARINC\\_STATUS](#) structure, which will be filled with status information.

### RESULT

Success: ARINC\_SUCCESS

Error: NTCAN\_INVALID\_HANDLE - An invalid handle

NTCAN\_INVALID\_PARAMETER - pStatus is NULL

### SEE ALSO

[CAN-errors](#)

[ARINC\\_STATUS](#)

[arincStatusReset\(\)](#)

### 7.6 arincStatusReset

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincStatusReset**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincStatusReset(ARINC_HANDLE hnd)
```

#### FUNCTION

All dynamic status information stored within [ARINC\\_STATUS](#) structure is reset (s. [ARINC\\_STATUS](#) description, fields are marked with (\*)).

#### PARAMETERS

hnd -- IN:  
A valid handle, which is associated with the CAN bus in question.

#### RESULT

Success: ARINC\_SUCCESS  
Error: NTCAN\_INVALID\_HANDLE - An invalid handle

#### SEE ALSO

[ARINC\\_STATUS](#)  
[arincStatus\(\)](#)

## 7.7 arincFormatError

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincFormatError**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincFormatError(ARINC_RESULT error,
                                                    UINT32          type,
                                                    char          *pBuf,
                                                    UINT32          bufsize)
```

### FUNCTION

Returns a string representation of the given error code.

### PARAMETERS

error -- IN:

An error code returned by any of the functions contained in this library,

type -- IN:

Select between two string representations, a rather short one and a verbose one (see [Errorformats](#), types are defined in [NTCAN](#)).

pBuf -- IN/OUT:

A pointer to a buffer, where the string is copied to.

Note:

If the buffer is too small, the string might get truncated.

bufsize -- IN:

Size of the target buffer.

### RESULT

Success: ARINC\_SUCCESS

Error: NTCAN\_INVALID\_PARAMETER - pBuf is NULL, bufsize is zero or  
type is no known output type

### SEE ALSO

[Errorformats](#)

[arincFormatEvent\(\)](#)

### 7.8 arincErrorHandler

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincErrorHandler**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincErrorHandler(ARINC_HANDLE      hnd,
                                                    ARINC_ERROR_HANDLER
pCallback,
                                                    void                    *pParam,
                                                    ARINC_RESULT
errorCode)
```

#### FUNCTION

This function can be used to register an error handler function, which will be called asynchronously on occurrence of certain error conditions in combination with ARINC825 scheduling.

#### NOTES

You can specify only one single error handler per [ARINC\\_HANDLE](#). The handler will only be triggered for the error codes listed below:

- [ARINC\\_CAN\\_STATE\\_CHANGE](#)
- [ARINC\\_CAN\\_ERROR](#)

For the error handler to work, scheduling needs to be started on the same handle.

#### PARAMETERS

**hnd -- IN:**  
A valid handle.

**pCallback -- IN:**  
Function pointer of type [ARINC\\_ERROR\\_HANDLER](#). Set to NULL to disable error handler.

**pParam -- IN:**  
A pointer size argument, which will be passed as user parameter together with other error information in [ARINC\\_ERROR](#) structure to the error handler.

**errorCode -- IN:**  
Specify an error code, you want the handler to be called for (to be chosen from the list of supported codes above). Set to zero, in order to have it called for all supported calls.

#### RESULT

Success: [ARINC\\_SUCCESS](#)  
Error: [NTCAN\\_INVALID\\_HANDLE](#) - An invalid handle

#### SEE ALSO

[CAN-errors](#)  
[ARINC\\_ERROR\\_HANDLER](#)  
[ARINC\\_ERROR](#)



## 7.9 arincTimeGet

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincTimeGet**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincTimeGet(ARINC_HANDLE hnd,
                                                UINT64          *pTime,
                                                UINT32          *pStatus)
```

### FUNCTION

Returns the current time and optionally its status.

### NOTES

If you have a special CAN hardware (for example with an IRIG-B receiver), the status of your time source will be returned via pStatus. In order to decode the status correctly please refer to the respective documentation of your IRIG-B hardware or of the accompanied library.

### PARAMETERS

hnd -- IN:

A valid handle of type [ARINC\\_HANDLE](#).

pTime -- OUT:

Pointer to a UINT64, wherein the current time will be stored (s. [ARINC-Time](#)).

pStatus -- OUT:

Pointer to a UINT32, wherein the status of time will be returned.

This may be set to NULL, if the status is of no interest or is not available.

### RESULT

Success: [ARINC\\_SUCCESS](#)

Error: [NTCAN\\_INVALID\\_HANDLE](#) - An invalid handle was used

[NTCAN\\_INVALID\\_PARAMETER](#) - pTime is NULL

### SEE ALSO

[ARINC-Time](#)

### 7.10 arincIntervalSet

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

NAME

**arincIntervalSet**

SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincIntervalSet(ARINC_HANDLE hnd,
                                                    UINT64           time,
                                                    UINT64           timeStart)
```

FUNCTION

Configures the ARINC825 scheduling interval and an optional start time.

NOTES

This function can not be called, when scheduling has already been started, in such case call [arincScheduleStop\(\)](#) or [arincTxStop\(\)](#) first.

Scheduling won't be activated by this function, regardless if the optional start time is used. Scheduling always needs to be activated by [arincScheduleStart\(\)](#) or [arincTxStart\(\)](#).

Beware:

Technically it is possible to call [arincIntervalSet\(\)](#) with one [ARINC\\_HANDLE](#), while scheduling has already been started by another. This will lead to undeterministic scheduling behaviour and is generally not advised.

PARAMETERS

**hnd** -- IN:  
A valid handle of type [ARINC\\_HANDLE](#).

**time** -- IN:  
Duration of a ARINC825 timeslice (see [ARINC-Time](#)).

**timeStart** -- IN:  
Time, when the first timeslice begins.  
If set to zero, the scheduling will begin immediately after [arincScheduleStart\(\)](#) or [arincTxStart\(\)](#) has been called.

RESULT

Success: [ARINC\\_SUCCESS](#)

Error: [NTCAN\\_INVALID\\_HANDLE](#) - An invalid handle was used  
[ARINC\\_ERROR\\_SCHED\\_ENABLED](#) - Scheduling needs to be stopped  
[ARINC\\_ERROR\\_TIME\\_NOT\\_SET](#) - Failed to configure interval or start time  
[ARINC\\_ERROR\\_INTERVAL\\_LOW](#) - time too small (below 1ms)

SEE ALSO

[ARINC-Time](#)  
[arincIntervalGet\(\)](#)

## 7.11 arincIntervalGet

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincIntervalGet**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincIntervalGet(ARINC_HANDLE hnd,
                                                    UINT64          *pTime,
                                                    UINT64          *pTimeStart)
```

### FUNCTION

Reads the currently configured ARINC825 scheduling interval.

### NOTES

In \*pTimeStart the start of the next timeslice is returned. This is equal to timeStart configured with [arincIntervalSet\(\)](#) as long as scheduling has not started, yet. When the start time has passed, the start of the next timeslice is returned.

### PARAMETERS

hnd -- IN:

A valid handle of type [ARINC\\_HANDLE](#).

pTime -- OUT:

Pointer to UINT64, wherein the currently configured interval is returned.

pTimeStart -- OUT:

Pointer to UINT64, wherein the configured/current start time of the next timeslice is returned.

### RESULT

Success: ARINC\_SUCCESS

Error: NTCAN\_INVALID\_HANDLE - An invalid handle was used  
NTCAN\_INVALID\_PARAMETER - pTime or pTimeStart is NULL

### SEE ALSO

[ARINC-Time](#)

[arincIntervalSet\(\)](#)

### 7.12 arincRxStart

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincRxStart**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincRxStart(ARINC_HANDLE hnd,
                                                INT32          prioMode,
                                                INT32          prioRx)
```

#### FUNCTION

Starts the RX daemon thread independent of the TX daemon thread.

#### NOTES

Alternatively, use **arincRxStart()** and **arincTxStart** instead of **arincScheduleStart()**.

#### PARAMETERS

hnd -- IN:

A valid handle of type **ARINC\_HANDLE**.

prioMode -- IN:

Use one of the **Thread-Priority** defines to choose between high priority or priority inheritance (both modes system independent) or manual configuration of system dependent priorities.

prioRx -- IN:

Specify the priority of the RX daemon thread.

This parameter is used only, if prioMode is set to **ARINC\_PRIO\_SET**.

The priority value is system specific. Special care needs to be taken to write a system independent application.

#### RESULT

Success: **ARINC\_SUCCESS**

Error: **ARINC\_INVALID\_HANDLE** - An invalid handle was used  
**ARINC\_INVALID\_PARAMETER** - Invalid value for prioMode or prioRx, prioTx are out of range for current host system

#### SEE ALSO

[arincRxStop\(\)](#)  
[arincTxStart\(\)](#)  
[arincTxStop\(\)](#)  
[arincScheduleStart\(\)](#)  
[arincScheduleStop\(\)](#)  
[Thread-Priority](#)

## 7.13 arincRxStop

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincRxStop**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincRxStop(ARINC_HANDLE hnd)
```

### FUNCTION

Deactivates the RX daemon thread.

### NOTES

Alternatively, use **arincRxStop**() and [arincTxStop](#) instead of [arincScheduleStop](#)().

### PARAMETERS

hnd -- IN:  
A valid handle of type [ARINC\\_HANDLE](#).

### RESULT

Success: ARINC\_SUCCESS  
Error: ARINC\_INVALID\_HANDLE - An invalid handle was used

### SEE ALSO

[arincRxStart](#)()  
[arincTxStart](#)()  
[arincTxStop](#)()  
[arincScheduleStart](#)()  
[arincScheduleStop](#)()

### 7.14 arincTxStart

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincTxStart**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincTxStart(ARINC_HANDLE hnd,  
                                                INT32 prioMode,  
                                                INT32 prioTx)
```

#### FUNCTION

After a scheduling table has been defined (using [arincObjAdd\(\)](#)) and time slice duration has been configured (using [arincIntervalSet\(\)](#)), the actual TX scheduling is activated with this function.

#### NOTES

Even if a start time has been configured with [arincIntervalSet\(\)](#) it is still needed to call **arincTxStart()**.

If you want to develop an application, which non intrusively works within a schedule configured by another application, you can use [arincIntervalGet\(\)](#) to gather the needed timing information on this handle and avoid the otherwise needed [arincIntervalSet\(\)](#) call.

Alternatively, use [arincRxStart\(\)](#) and **arincTxStart** instead of [arincScheduleStart\(\)](#).

#### PARAMETERS

hnd -- IN:

A valid handle of type [ARINC\\_HANDLE](#).

prioMode -- IN:

Use one of the [Thread-Priority](#) defines to choose between high priority or priority inheritance (both modes system independent) or manual configuration of system dependent priorities.

prioTx -- IN:

Specify the priority of the TX daemon thread.

This parameter is used only, if prioMode is set to [ARINC\\_PRIO\\_SET](#).

The priority value is system specific. Special care needs to be taken to write a system independent application.

#### RESULT

Success: [ARINC\\_SUCCESS](#)

Error: [ARINC\\_INVALID\\_HANDLE](#) - An invalid handle was used  
[ARINC\\_INVALID\\_PARAMETER](#) - Invalid value for prioMode or prioRx,  
prioTx are out of range for current host  
system  
[ARINC\\_ERROR\\_NO\\_INTERVAL](#) - Scheduling can not be started,  
because there's no valid timeslice  
interval configured, call  
[arincIntervalSet\(\)](#) before  
**arincTxStart()**

#### SEE ALSO

[arincRxStart\(\)](#)

[arincRxStop\(\)](#)

arincTxStop ()  
arincScheduleStart ()  
arincScheduleStop ()  
arincIntervalSet ()  
arincIntervalGet ()  
Thread-Priority

### 7.15 arincTxStop

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincTxStop**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincTxStop(ARINC_HANDLE hnd)
```

#### FUNCTION

Deactivates TX scheduling.

#### NOTES

If reconfiguration of scheduling table is needed, this function needs to be called first.

Alternatively, use [arincRxStop\(\)](#) and **arincTxStop()** instead of [arincScheduleStop\(\)](#).

#### PARAMETERS

hnd -- IN:  
A valid handle of type [ARINC\\_HANDLE](#).

#### RESULT

Success: ARINC\_SUCCESS  
Error: ARINC\_INVALID\_HANDLE - An invalid handle was used

#### SEE ALSO

[arincRxStart\(\)](#)  
[arincRxStop\(\)](#)  
[arincTxStart\(\)](#)  
[arincScheduleStart\(\)](#)  
[arincScheduleStop\(\)](#)



## 7.16 arincScheduleStart

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincScheduleStart**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincScheduleStart(ARINC_HANDLE hnd,
                                                    INT32          prioMode,
                                                    INT32          prioRx,
                                                    INT32          prioTx)
```

### FUNCTION

After a scheduling table has been defined (using [arincObjAdd\(\)](#)) and time slice duration has been configured (using [arincIntervalSet\(\)](#)), the actual scheduling is activated with this function.

### NOTES

Even if a start time has been configured with [arincIntervalSet\(\)](#) it is still needed to call [arincScheduleStart\(\)](#).

If you want to develop an application, which non intrusively works within a schedule configured by another application, you can use [arincIntervalGet\(\)](#) to gather the needed timing information on this handle and avoid the otherwise needed [arincIntervalSet\(\)](#) call.

Alternatively, use [arincRxStart\(\)](#) and [arincTxStart](#) instead of [arincScheduleStart\(\)](#).

### PARAMETERS

hnd -- IN:

A valid handle of type [ARINC\\_HANDLE](#).

prioMode -- IN:

Use one of the [Thread-Priority](#) defines to choose between high priority or priority inheritance (both modes system independent) or manual configuration of system dependent priorities.

prioRx -- IN:

Specify the priority of the RX daemon thread.

This parameter is used only, if prioMode is set to [ARINC\\_PRIO\\_SET](#).

The priority value is system specific. Special care needs to be taken to write a system independent application.

prioTx -- IN:

Specify the priority of the TX daemon thread.

This parameter is used only, if prioMode is set to [ARINC\\_PRIO\\_SET](#).

The priority value is system specific. Special care needs to be taken to write a system independent application.

### RESULT

Success: [ARINC\\_SUCCESS](#)

Error: [NTCAN\\_INVALID\\_HANDLE](#) - An invalid handle was used  
[NTCAN\\_INVALID\\_PARAMETER](#) - Invalid value for prioMode or prioRx, prioTx are out of range for current host system  
[ARINC\\_ERROR\\_NO\\_INTERVAL](#) - Scheduling can not be started, because there's no valid timeslice interval configured, call

## Functions

---

`arincIntervalSet()` before  
`arincScheduleStart()`

### SEE ALSO

`arincScheduleStop()`  
`arincRxStart()`  
`arincRxStop()`  
`arincTxStart()`  
`arincTxStop()`  
`arincIntervalSet()`  
`arincIntervalGet()`  
`Thread-Priority`

## 7.17 arincScheduleStop

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincScheduleStop**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincScheduleStop(ARINC_HANDLE hnd)
```

### FUNCTION

Deactivates scheduling.

### NOTES

If reconfiguration of scheduling table is needed, this function needs to be called first.

Alternatively, use [arincRxStop\(\)](#) and [arincTxStop\(\)](#) instead of **arincScheduleStop()**.

### PARAMETERS

hnd -- IN:  
A valid handle of type [ARINC\\_HANDLE](#).

### RESULT

Success: [ARINC\\_SUCCESS](#)  
Error: [NTCAN\\_INVALID\\_HANDLE](#) - An invalid handle was used

### SEE ALSO

[arincScheduleStart\(\)](#)  
[arincRxStart\(\)](#)  
[arincRxStop\(\)](#)  
[arincTxStart\(\)](#)  
[arincTxStop\(\)](#)

### 7.18 arincPoll

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

#### NAME

**arincPoll**

#### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincPoll(ARINC_HANDLE hnd,  
                                             ARINC_CMSG_T *pCmsg,  
                                             INT32 *pNum)
```

#### FUNCTION

By means of **arincPoll()** the current state of any ARINC825 object (regardless, if TX or RX object) can be polled. The state consists of the number of valid data bytes as well as currently contained data bytes, timestamp of last reception/transmission and transmission and reception counters. The length field also contains information, if the object has received any data yet (ARINC\_NO\_DATA) or if the data has been updated since last call of **arincPoll()** (ARINC\_OLD\_DATA).

#### PARAMETERS

hnd -- IN:

A valid handle of type [ARINC\\_HANDLE](#).

pCmsg -- IN/OUT:

Pointer to one or more [ARINC\\_CMSG\\_T](#) structures. These have to be initialized with the CAN IDs of the ARINC objects in question prior to calling **arincPoll()**.

pNum -- IN/OUT:

Pointer to an INT32, which determines the number of objects pCmsg is pointing to (and thus the number of objects to poll).

The value of polled objects (this might be lower, than the number requested, if an error occurred) is returned via pNum.

#### RESULT

Success: ARINC\_SUCCESS

Error: NTCAN\_INVALID\_HANDLE - An invalid handle was used  
NTCAN\_INVALID\_PARAMETER - Either pCmsg and/or pNum is NULL

#### SEE ALSO

[arincObjAdd\(\)](#)  
[arincObjDelete\(\)](#)  
[arincTxObjUpdate\(\)](#)

## 7.19 arincTxObjUpdate

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincTxObjUpdate**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincTxObjUpdate(ARINC_HANDLE hnd,
                                                    ARINC_CMSG_T *pCmsg,
                                                    INT32 *pNum)
```

### FUNCTION

Updates the data of one or more objects formerly added by `arincObjectAdd()`.

### NOTES

This function works only for ARINC825 "transmit" objects (group >= 0) and not on ARINC objects of ARINC\_GROUP\_RX.

### PARAMETERS

`hnd` -- IN:

A valid handle of type `ARINC_HANDLE`.

`pCmsg` -- IN:

Pointer to one or more `ARINC_CMSG_T` structures. These have to contain the number of data bytes as well as the bytes themselves and the CAN ID as reference of the objects (other parts of `ARINC_CMSG_T` structure are ignored).

`pNum` -- IN/OUT:

Pointer to an INT32, which determines the number of objects `pCmsg` is pointing to (and thus the number of objects to update).

The value of updated objects (this might be lower, than the number requested, if an error occurred) is returned via `pNum`.

### RESULT

Success: `ARINC_SUCCESS`

Error: <code>NTCAN_INVALID_HANDLE</code>	- An invalid handle was used
<code>NTCAN_INVALID_PARAMETER</code>	- Either <code>pCmsg</code> and/or <code>pNum</code> is NULL
<code>ARINC_ERROR_ID_NOT_FOUND</code>	- The CAN ID of a given object was not found. Call <code>arincObjAdd()</code> first.
<code>ARINC_ERROR_NOT_TX</code>	- A given object was not configured as a TX object (group >= 0)

### SEE ALSO

`arincObjAdd()`  
`arincObjDelete()`  
`arincPoll()`

## 7.20 arincTxObjDisable

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincTxObjDisable**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincTxObjDisable(ARINC_HANDLE hnd,
                                                    ARINC_CMSG_T *pCmsg,
                                                    INT32 *pNum,
                                                    INT32 flag)
```

### FUNCTION

Disables or enables an ARINC825 "transmit" object. If disabled this object won't be transmitted any longer.

### NOTES

This function works only for ARINC825 "transmit" objects (group >= 0). On ARINC825 objects of ARINC\_GROUP\_RX the call has no effect.

By default a newly created ARINC object is enabled.

(De-)activation of ARINC825 objects (even if done within one call for more than one object) is NOT atomic. Also there's no guarantee, the given objects are dis-/enabled within one timeslot. This is done to prevent any disturbance of the scheduling, even if large amounts of objects are dis-/enabled.

### PARAMETERS

hnd -- IN:

A valid handle of type [ARINC\\_HANDLE](#).

pCmsg -- IN:

Pointer to one or more [ARINC\\_CMSG\\_T](#) structures. These have to contain the CAN ID as reference of the objects (other parts of [ARINC\\_CMSG\\_T](#) structure are ignored).

pNum -- IN/OUT:

Pointer to an INT32, which determines the number of objects pCmsg is pointing to (and thus the number of objects to dis-/enable).

The value of dis-/enabled objects (this might be lower, than the number requested, if an error occurred) is returned via pNum.

flag -- IN:

Setting flag true (unequal zero) disables the referenced ARINC825 objects. Setting this flag zero, reenables the objects.

### RESULT

Success: [ARINC\\_SUCCESS](#)

Error: [NTCAN\\_INVALID\\_HANDLE](#) - An invalid handle was used  
[NTCAN\\_INVALID\\_PARAMETER](#) - Either pCmsg and/or pNum is NULL  
[ARINC\\_ERROR\\_ID\\_NOT\\_FOUND](#) - The CAN ID of a given object was not found. Call [arincObjAdd\(\)](#) first.

### SEE ALSO

[arincObjAdd\(\)](#)  
[arincObjDelete\(\)](#)  
[arincTxObjUpdate\(\)](#)

## 7.21 arincWaitForTimeslot

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincWaitForTimeslot**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincWaitForTimeslot(ARINC_HANDLE hnd,  
                                                       INT32 timeout)
```

### FUNCTION

Function returns, when a new timeslice begins, or when timeout expired.

### NOTES

Scheduling needs to be started, before this function is called.

### PARAMETERS

hnd -- IN:  
A valid handle of type [ARINC\\_HANDLE](#).  
timeout -- IN:  
Timeout in milliseconds.

### RESULT

Success: ARINC\_SUCCESS  
Error: NTCAN\_INVALID\_HANDLE - An invalid handle was used  
ARINC\_ERROR\_SCHED\_DISABLED - Function called without scheduling enabled  
NTCAN\_RX\_TIMEOUT - Timeout expired without receiving a new timeslice event from hardware. Either this happens on purpose (e.g. due to small timeouts) or there's a severe problem in scheduling.

### 7.22 arincObjAdd

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

NAME

**arincObjAdd**

SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincObjAdd(ARINC_HANDLE hnd,
                                               ARINC_CMSG_T *pCmsg,
                                               INT32          *pNum)
```

FUNCTION

Add one or more objects to an ARINC825 scheduling table. The number of objects pointed to by pCmsg has to be specified in pNum. Every object has ARINC825 attributes, such as group, column (m) and slice index (n). By setting group to ARINC\_GROUP\_RX a receive object is added (in this case m and n are ignored).

NOTES

Scheduling needs to be stopped, before this function is called.

Objects are referenced by their [CAN-IDs](#). Each CAN-ID can be added once, only. pNum returns the number of successfully added objects. Normally this value shouldn't change, but in case of an error it might be used to determine, which object was cause of the error. Once an object was added, it won't be removed, if an error occurs with one of the subsequent objects.

PARAMETERS

hnd -- IN:  
A handle of type [ARINC\\_HANDLE](#).  
pCmsg -- IN:  
Pointer to one or more [ARINC\\_CMSG\\_T](#), which will be added to the schedule.  
pNum -- IN/OUT:  
Pointer to a INT32, which determines the number of objects pCmsg is pointing to. When returning, it contains the number of successfully added objects.

RESULT

Success:	ARINC_SUCCESS	
Error:	NTCAN_INVALID_HANDLE	- An invalid handle was used
	NTCAN_INVALID_PARAMETER	- Either pCmsg or pNum was NULL, or one of the objects contained invalid attributes (e.g. an invalid ARINC group was specified or "n" didn't fit group) Use pNum to find the object
	ARINC_ERROR_SCHED_ENABLED	- <a href="#">arincScheduleStart()</a> or <a href="#">arincTxStart()</a> has been called before. Scheduling needs to be stopped, before adding new objects
	NTCAN_INSUFFICIENT_RESOURCES	- Not enough memory to add another object
	ARINC_ERROR_ID_BUSY	- The CAN ID has already been added
	ARINC_ERROR_COL_BUSY	- The column with the specified group and slice index is already occupied

SEE ALSO

[ARINC\\_CMSG\\_T](#)  
[arincObjDelete\(\)](#)



## 7.23 arincObjDelete

[ [Top](#) ] [ [Functions](#) ] [ [Functions](#) ]

### NAME

**arincObjDelete**

### SYNOPSIS

```
EXPORT ARINC_RESULT PSYS_CALLTYPE arincObjDelete(ARINC_HANDLE hnd,
                                                  ARINC_CMSG_T *pCmsg,
                                                  INT32          *pNum)
```

### FUNCTION

Delete one or more objects from an ARINC825 scheduling table. The number of objects pointed to by pCmsg has to be specified in pNum. The ARINC825 attributes, such as group, column (m) and slice index (n) are ignored by this call.

### NOTES

Scheduling needs to be stopped, before this function is called.

Objects are referenced by their CAN ID. pNum returns the number of successfully deleted objects. Normally this value shouldn't change, but in case of an error it might be used to determine, which object was cause of the error. Once an object was deleted, it won't be readded, if an error occurs with one of the subsequent objects.

### PARAMETERS

hnd -- IN:  
A handle of type [ARINC\\_HANDLE](#).

pCmsg -- IN:  
Pointer to one or more [ARINC\\_CMSG\\_T](#), which will be deleted from schedule.

pNum -- IN/OUT:  
Pointer to an INT32, which determines the number of objects pCmsg is pointing to. When returning, it contains the number of successfully deleted objects.

### RESULT

Success:	ARINC_SUCCESS	
Error:	NTCAN_INVALID_HANDLE	- An invalid handle was used
	NTCAN_INVALID_PARAMETER	- Either pCmsg or pNum was zero, or one of the objects contained invalid attributes. Use pNum to find the object.
	ARINC_ERROR_SCHED_ENABLED	- <a href="#">arincScheduleStart()</a> or <a href="#">arincTxStart()</a> has been called before. Scheduling needs to be stopped, before deleting objects.
	ARINC_ERROR_ID_NOT_FOUND	- The CAN ID has not been previously added.

### SEE ALSO

[ARINC\\_CMSG\\_T](#)  
[arincObjAdd\(\)](#)

## 8. ARINC825 LabVIEW Library

### INTRODUCTION

The contained VI's provide you with means to make use of ARINC825 timeslice scheduling on esd CAN hardware. In general there are two sets of VI's.

One set, called A825 VIs (signal based VIs), provides a project file based signal approach, probably most suitable for most LabVIEW users.

The other set, called ARINC VIs (native VIs), offers direct access to the entire ARINC825 library, tailored for the experienced user with programming knowledge, who wants to have control over every detail.

### REQUIREMENTS

LabVIEW Version:	LabVIEW 2013 or later
CAN Interface:	esd CAN Interface, best with esd 400 family with IRIG-B Support (e.g. PMC-CAN/400-4 IRIG-B)
CAN Driver:	esd CAN Driver with NTCAN Library Support
ARINC825 Library:	Version 1.1.15 or later
IRIG-B LabVIEW Library (only with CAN Interface with IRIG-B):	Version 13.1.1 or later

## 8.1 Archive contents

The LabVIEW\_ARINC825\_VERSION.zip archive contains following folders:

```
.
|-- ARINC825_LabVIEW_Examples
`-- user.lib
    |-- LabVIEW_can_arinc825_esd
        |-- SubVI
        |-- VIs
```

## **8.2 Installation**

- Install CAN SDK from ARINC825 CD or CAN CD  
(Select IRIG-B Option if you have a CAN interface with IRIG-B option)
- Install ARINC 825 Library from ARINC825 CD  
Copy the `user.lib` folder from `LabVIEW_ARINC825_VERSION.zip` into your LabVIEW installation.

## 8.3 Basic Usage Information

In order to use either set of VI's, you will first need to acquire a handle, which needs to be passed to all successive VI's. There are two different handles, one for each set of VI's.

The ARINC (native) handle is returned from "nativeArincHandleOpen.vi" and can be used with the ARINC VI's only.

The A825 (also called "project handle") is returned from "A825ProjectOpen.vi". It is advised to use this project handle with the set of A825 VI's, nevertheless it can also be used with the set of ARINC VI's.

For further information on general functionality of ARINC825 and/or every single VI please have a look at one or more of the following references:

- Context help of the VI's inside of LabVIEW (probably the easiest and most convenient option)
- Examples included within the archive may give good starting points
- HTML documentation of esd's ARINC825 library (included with the library)
- esd's NTCAN API documentation for further information on CAN (<http://esd.eu/en/products/can-sdk>)
- ARINC825 specification (<http://esd.eu/en/manuals/arinc825-software-manual>)

## 8.4 A825 VIs (Signal Based VIs)

The A825 VIs (Signal Based VIs) provide a project file based signal approach, probably most suitable for all LabVIEW users.

This VIs are named with the prefix “A825”. Their VI icons look like this (e.g.):



OVERVIEW OF A825 VIs:

[A825Error2String](#)

[A825Info](#)

[A825ObjectPoll](#)

[A825ObjectSend](#)

[A825ObjectTrigger](#)

[A825ProjectClose](#)

[A825ProjectOpen](#)

[A825ScheduleStart](#)

[A825ScheduleStop](#)

[A825SignalPoll](#)

[A825SignalTrigger](#)

[A825SignalUpdate](#)

[A825Status](#)

[A825StatusReset](#)

[A825TimeGet](#)

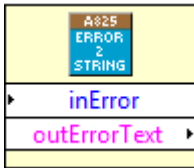
[A825TxObjectDisable](#)

[A825TxObjectUpdate](#)

The parameters of the A825 VIs are described in the online help. Click with the right mouse button on a parameter to open the context menu which contains further information about this parameter.

An example of a project file is given in chapter: “Example of a Project File”, page 70.

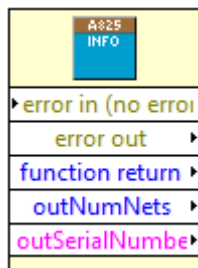
## 8.4.1 A825Error2String



Convert any error code from the ARINC825 VI's into a human readable string.

```
EXPORT int CALLTYPE A825Error2String(const int inError, char * const outErrorText);
```

## 8.4.2 A825Info



Provides information about the esd CAN hardware in the system.

```
EXPORT int CALLTYPE A825Info(int32_t * const outNumNets, char * outSerialNumbers);
```



### 8.4.3 A825ObjectPoll

A825 OBJECT POLL	
▶ error in (no error)	
▶ inCanId	
▶ inHndProject	
error out	▶
function return	▶
outData	▶
outFlags	▶
outLen	▶
outTime	▶


Retrieve the current state of an ARINC825 object, referenced by a CAN ID. This may be an RX as well as TX object.

Note:

One ARINC825 object may contain multiple signals.

```
EXPORT int CALLTYPE A825ObjectPoll(const uint32_t inHndProject, const int32_t inCanId, uint8_t* const outLen, uint64_t* const outData, uint32_t* const outFlags, uint64_t* const outTime);
```

## 8.4.4 A825ObjectSend



▶ error in (no error)
▶ inCanId
▶ inData
▶ inHndProject
▶ inLen
error out ▶
function return ▶

Manually send an ARINC825 object (identical to CAN frame in this case).

If there's an ARINC825 object with the same CAN ID defined in the project, this object will get updated in the process.

```
EXPORT int CALLTYPE A825ObjectSend(const uint32_t inHndProject, const int32_t inCanId, const uint8_t inLen, const uint64_t inData);
```

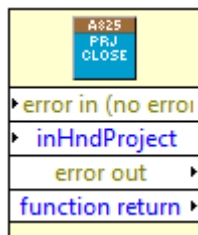
## 8.4.5 A825ObjectTrigger


▶ error in (no error)
▶ inCanId
▶ inHndProject
error out ▶
function return ▶

Manually trigger the transmission of an ARINC825 object in its current state.

```
EXPORT int CALLTYPE A825ObjectTrigger(const uint32_t inHndProject, const int32_t inCanId);
```

## 8.4.6 A825ProjectClose

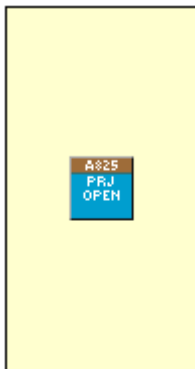


Close a project, formerly opened with "A825 Project Open.vi".

Note: This has to be called BEFORE stopping your LabView application, in order to assure a correct cleanup!

```
EXPORT int CALLTYPE A825ProjectClose(const uint32_t inHndProject);
```

## 8.4.7 A825ProjectOpen



error in (no error)
inFilename
inoutBaud
inoutNet
inoutTimeInterv
inoutTimeStart
error out
function return
inoutBaud out
inoutNet out
inoutTimeInterv
inoutTimeStart
outArinc825Libr
outBoardId
outBoardstatus
outCanCtrlType
outCanLibraryVer
outDriverVersior
outFeatures
outFirmwareVer
outHardwareVer
outHndProject
outNumObjects
outNumSignals
outSerialNumbe
outTimeFreq

Open a project file and return a project handle as reference for all successive ARINC825 VIs.  
The various inputs may be used to override certain parameters of the specified project file.

Note: In contrast to the ARINC825-C-API "A825 Project Open.vi" provides some static information, normally provided by arincStatus().  
This is done for the users convenience and should be selfexplanatory in its use.

```
EXPORT int CALLTYPE A825ProjectOpen(const char * const inFilename, int * const inoutNet, uint32_t * const inoutBaud, uint64_t * const outTimeFreq, uint64_t * const inoutTimeStart, uint64_t * const inoutTimeInterval, uint32_t * const outNumObjects, uint32_t * const outNumSignals, uint32_t * const inoutIrigBInput, uint32_t * const inoutIrigBMode, uint32_t * const outHndProject);
```

### 8.4.7.1 Example of a Project File

The VI A825ProjectOpen opens a project file. For example:

```
# LabView ARINC825 project file
#
# Comment lines are allowed everywhere and have to be preceded by a hash character '#'
#
# The project file needs at least a "ARINC825Config" section.
# Sections are defined by section name in square brackets ("[" , "]" ).
#
# There should be only one project file per CAN bus.
#
# All project files on the same CAN board should have the same IRIG-B configuration,
# otherwise the last one loaded will take effect.
#
# Two more sections may be used:
# "ARINC825Objects" to define ARINC825/CAN objects
# "ARINC825Signals" to map LabView signal names into CAN objects
#
# Numerical values may be specified in decimal or hexadecimal (beginning with "0x") notation.
#
# To separate values, space as well as tabs may be used.
#

[ARINC825Config]
# Following values may be set (omitted values are set to default):
# Net (default: 0) - Number of CAN bus
# Baud (default: 0 (1MBit/s)) - Baud rate on CAN bus
# Timeslice (default: 0x0000000008000000 (0.5s)) - Duration of timeslice
# IRIGBInput (default: 0) - To choose input of IRIG-B signal
# (0: Analog Front, 1: Digital Front)
# IRIGBMode (default: 0) - To choose evaluation year in IRIB-B
# signal (0: No year information,
# 1: Year embedded in IRIG-B signal)
Net=0
Baud=0
Timeslice=0x1312D00

[ARINC825Objects]
# Each object is defined by CAN ID, ARINC 825 group (G), ARINC 825 column (M),
# ARINC 825 slice index (N), length of CAN frame (LEN) and optionally up to eight
# data bytes (D0-D7)

# At first three RX objects (set G = -1)...
# CANID G M N LEN D0 D1 D2 D3 D4 D5 D6 D7
0x20000100 -1 0 0 1
0x20000101 -1 0 0 8
0x20000102 -1 0 0 8


# ... and then three more TX objects
0x20000000 0 0 0 8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x20000001 1 0 0 8 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x20000002 2 0 0 4 0x00 0x00 0x00 0x00

[ARINC825Signals]
# NAME: Signal name, 16 characters at max (case insensitive, may obviously NOT begin with
'#')
# CANID: CAN ID of CAN object the signal is mapped to
# BitMin: Range [0..BitMax] (beginning with "byte 0, bit 0" up to "byte 7 bit 7")
# BitMax: Range [BitMin..63] (beginning with "byte 0 bit 0" up to "byte 7 bit 7")

# NAME CANID BitMin BitMax
Temperature1 0x20000000 0 15
Pressure1 0x20000000 16 31
Temperature2 0x20000000 32 47
```

Pressure2	0x20000000	48	63
CurrentX	0x20000001	0	31
CurrentY	0x20000001	32	63
ButtonA	0x20000002	24	24
ButtonB	0x20000002	25	25
ButtonC	0x20000002	26	26
ButtonD	0x20000002	27	27
SelectorS	0x20000002	0	7
LED1	0x20000100	0	0
LED2	0x20000100	1	1
LED3	0x20000100	2	2
LED4	0x20000100	3	3
#PositionX	0x20000101	0	63
#PositionY	0x20000102	0	63

## 8.4.8 A825ScheduleStart


▶ error in (no error)
▶ inHndProject
error out ▶
function return ▶


Start the ARINC825 timeslice scheduling, after your project has been loaded and everything is configured as needed.

Note: If you are using CAN hardware with special timestamp sources (e.g. IRIG-B), you need to assure, that your timebase is stable BEFORE starting the scheduler.

```
EXPORT int CALLTYPE A825ScheduleStart(const uint32_t inHndProject);
```




## 8.4.9 A825ScheduleStop


▶ error in (no error)
▶ inHndProject
error out ▶
function return ▶

Stop the ARINC825 timeslice scheduling. This is needed in order to perform configuration changes.

```
EXPORT int CALLTYPE A825ScheduleStop(const uint32_t inHndProject);
```


## 8.4.10 A825SignalPoll


▶ error in (no error)
▶ inHndProject
▶ inSignalname
error out ▶
function return ▶
outData ▶
outFlags ▶
outTime ▶

Get the current value of a signal defined in your project. The signal is referenced by its name.

```
EXPORT int CALLTYPE A825SignalPoll(const uint32_t inHndProject, const char *  
const inSignalname, uint64_t * const outData, uint32_t * const outFlags, uint64_t * const outTime);
```

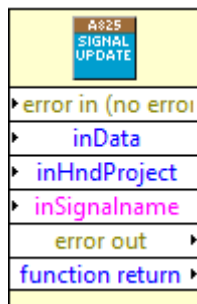
## 8.4.11 A825SignalTrigger


▶ error in (no error)
▶ inHndProject
▶ inSignalName
error out ▶
function return ▶

Triggers the transmission of the entire ARINC825 object the referenced signal is located in. Of course all other signals located within the same object are transmitted as well (obviously).

```
EXPORT int CALLTYPE A825SignalTrigger(const uint32_t inHndProject, const char * const inSignalName);
```

## 8.4.12 A825SignalUpdate




Change the value of a signal. Most commonly used on signals located in ARINC825 objects scheduled for transmission.

Note: This VI does not trigger the transmission itself.

```
EXPORT int CALLTYPE A825SignalUpdate(const uint32_t inHndProject, const char * const inSignalname, const uint64_t inData);
```

## 8.4.13 A825Status


	
▶	error in (no error)
▶	inHndProject
	error out ▶
	function return ▶
	outCanErrorCou▶
	outCanErrorCou▶
	outCanStatus ▶
	outErrorCode ▶
	outErrorCodeTir▶
	outErrorCount ▶
	outErrorLast ▶
	outErrorLastTim▶
	outErrorLostFrar▶
	outRxCount ▶
	outRxLastTime ▶
	outTime ▶
	outTxCount ▶
	outTxLastTime ▶

Provides a bunch of status information.

Note: In contrast to the ARINC825-C-API "A825 Status.vi" provides information subject to change, instead of providing all infos normally provided by arincStatus(). This is done for the users convenience and should be selfexplanatory in its use. The static information is to be found as output of "A825 Project Open.vi".

```
EXPORT int CALLTYPE A825Status(const uint32_t inHndProject, uint64_t * const outTime, uint32_t * const outRxCount, uint64_t * const outRxLastTime, uint32_t * const outTxCount, uint64_t * const outTxLastTime, uint32_t * const outErrorCount, int32_t * const outErrorLast, uint64_t * const outErrorLastTime, uint32_t * const outErrorCode, uint64_t * const outErrorCodeTime, uint8_t * const outCanStatus, uint8_t * const outCanErrorCountRx, uint8_t * const outCanErrorCountTx, uint32_t * const outErrorLostFrames);
```

## 8.4.14 A825StatusReset


▶ error in (no error)
▶ inHndProject
error out ▶
function return ▶


Resets the information delivered by "A825 Status.vi", such as error counters, TX- and RX-frame counters.

**Note:**

The CAN RX- and TX-error counters will not be reset by this function. This is not even possible with most CAN controllers. With

```
EXPORT int CALLTYPE A825StatusReset(const uint32_t inHndProject);
```

## 8.4.15 A825TimeGet


▶ error in (no error)
▶ inHndProject
error out ▶
function return ▶
outTime ▶
outTimeStatus ▶

Get current timestamp.

```
EXPORT int CALLTYPE A825TimeGet(const uint32_t inHndProject, uint64_t* const outTime, uint32_t* const outIrigBStatus);
```

## 8.4.16 A825TxObjectDisable

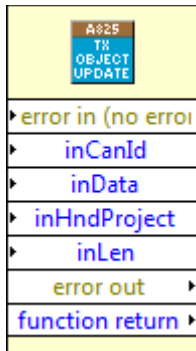
<b>A825 TX OBJECT DISABLE</b>
▶ error in (no error)
▶ inCanId
▶ inFlagDisable
▶ inHndProject
error out ▶
function return ▶

While ARINC825 scheduling is in process, this VI can be used to temporarily disable an ARINC825 object, meaning it will not be scheduled, unless it is reenabled again with this VI.

```
EXPORT int CALLTYPE A825TxObjectDisable(const uint32_t inHndProject, const int32_t inCanId, const int32_t inFlagDisable);
```



## 8.4.17 A825TxObjectUpdate



Update the contents of an ARINC825 object.

Note: This VI does not trigger the transmission itself.

```
EXPORT int CALLTYPE A825TxObjectUpdate(const uint32_t inHndProject, const int32_t inCanId, const uint8_t inLen, const uint64_t inData);
```

### 8.5 ARINC VIs (Native VIs)

The ARINC VIs (Native VIs) offer direct access to the entire ARINC825 library, tailored for the experienced user with programming knowledge, who wants to have control over every detail.

These VIs are named with the prefix “ARINC”. Their VI icons look like this (e.g.):



OVERVIEW OF ARINC VIs:

[ArincBaudrateGet](#)

[ArincBaudrateSet](#)

[ArincClose](#)

[ArincError2String](#)

[ArincHandleOpen](#)

[ArincIntervalGet](#)

[ArincIntervalSet](#)

[ArincObjAdd](#)

[ArincObjDelete](#)

[ArincPoll](#)

[ArincRxStart](#)

[ArincRxStop](#)

[ArincScheduleStart](#)

[ArincScheduleStop](#)

[ArincStatus](#)

[ArincStatusReset](#)

[ArincTimeGet](#)

[ArincTxObjDisable](#)

[ArincTxObjUpdate](#)

[ArincTxStart](#)

[ArincTxStop](#)

[ArincWaitForTimeslot](#)

## 8.5.1 ArincBaudrateGet

ARINC BAUD RATE GET
▶ error in (no error)
▶ inHnd
error out ▶
function return ▶
outBaud ▶

The "native" VI set directly calls ARINC825 library functions.

Please refer to the documentation of arincBaudrateGet().

```
EXPORT int CALLTYPE nativeArincBaudrateGet(const uint32_t inHnd, const uint32_t* outBaud);
```

See page 36 for the ARINC 825 library function description.

## 8.5.2 ArincBaudrateSet

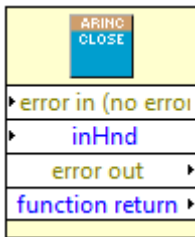
ARINC BAUD RATE SET
▶ error in (no error)
▶ inBaud
▶ inHnd
error out ▶
function return ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincBaudrateSet().

```
EXPORT int CALLTYPE nativeArincBaudrateSet(const uint32_t inHnd, const uint32_t inBaud);
```

See page 35 for the ARINC 825 library function description.

### 8.5.3 ArincClose

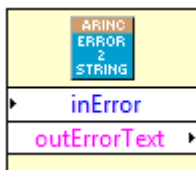


The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of `arincClose()`.

```
EXPORT int CALLTYPE nativeArincClose(const uint32_t inHnd);
```

See page 34 for the ARINC 825 library function description.

## 8.5.4 ArincError2String




Convert any error code from the ARINC825 VI's into a human readable string.

```
EXPORT int CALLTYPE nativeArincError2String(const int inError, char * const outErrorText);
```

See page 39 for the ARINC 825 library function description.

## 8.5.5 ArincHandleOpen

	
error in (no error)	▶
inNet	▶
error out	▶
function return	▶
outArinc825Libr	▶
outBoardId	▶
outBoardstatus	▶
outCanCtrlType	▶
outCanLibraryVe	▶
outDriverVersior	▶
outFeatures	▶
outFirmwareVer	▶
outFreq	▶
outHardwareVer	▶
outHnd	▶
outSerialNumbe	▶

The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of arincHandleOpen().

Note: This VI provides some more outputs than the arincHandleOpen() library call. The ARINC825 library delivers the same information via arincStatus().

```
EXPORT int CALLTYPE nativeArincHandleOpen(const uint32_t inNet, uint32_t * const outHnd, uint64_t * const outFreq, uint16_t * const outHardwareVersion, uint16_t * const outFirmwareVersion,
uint16_t * const outDriverVersion, uint16_t * const outCanLibraryVersion, uint32_t * const outBoardstatus, char * const outBoardId, char * const outSerialNumber, uint16_t * const outFeatures,
uint16_t * const outArinc825LibraryVersion, uint8_t * const outCanCtrlType);
```

See page 33 for the ARINC 825 library function description.

## 8.5.6 ArincIntervalGet

ARINC INTER VAL GET
▶ error in (no error)
▶ inHnd
error out ▶
function return ▶
outTime ▶
outTimeStart ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincIntervalGet().

```
EXPORT int CALLTYPE nativeArincIntervalGet(const uint32_t inHnd, uint64_t * const outTime, uint64_t * const outTimeStart);
```

See page 43 for the ARINC 825 library function description.



## 8.5.7 ArincIntervalSet

ARINC INTER VAL SET
▶ error in (no error)
▶ inHnd
▶ inTime
▶ inTimeStart
error out ▶
function return ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincIntervalSet().

**Note:**

May only be used, while scheduling is NOT started.

```
EXPORT int CALLTYPE nativeArincIntervalSet(const uint32_t inHnd, const uint64_t inTime, const uint64_t inTimeStart);
```

See page 42 for the ARINC 825 library function description.

## 8.5.8 ArincObjAdd

ARINC OBJ ADD
▶ error in (no error)
▶ inARINC_CMSG
▶ inHnd
error out ▶
function return ▶
inoutNum out ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincObjectAdd().

```
EXPORT int CALLTYPE nativeArincObjAdd(const uint32_t inHnd, ARINC_CMSG_T * const inCmsg, int32_t * const inoutNum);
```

See page 56 for the ARINC 825 library function description.

## 8.5.9 ArincObjDelete

ARINC OBJ DELETE
▶ error in (no error)
▶ inARINC_CMSG
▶ inHnd
error out ▶
function return ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincObjectDelete().

```
EXPORT int CALLTYPE nativeArincObjDelete(const uint32_t inHnd, ARINC_CMSG_T * const inCmsg, int32_t * const inoutNum);
```

See page 57 for the ARINC 825 library function description.

## 8.5.10 ArincPoll

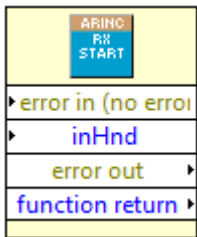
ARINC POLL
▶ error in (no error)
▶ inARINC_CMSG
▶ inHnd
error out ▶
function return ▶
inoutNum out ▶
outARINC_CMSG ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincPoll().

```
EXPORT int CALLTYPE nativeArincPoll(const uint32_t inHnd, ARINC_CMSG_T * const inoutCmsg, int32_t * const inoutNum);
```

See page 52 for the ARINC 825 library function description.

## 8.5.11 ArincRxStart

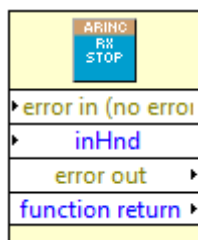


The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of `arincRxStart()`.

```
EXPORT int CALLTYPE nativeArincRxStart(const uint32_t inHnd);
```

See page 44 for the ARINC 825 library function description.

## 8.5.12 ArincRxStop

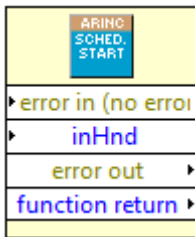


The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of arincRxStop().

```
EXPORT int CALLTYPE nativeArincRxStop(const uint32_t inHnd);
```

See page 45 for the ARINC 825 library function description.

## 8.5.13 ArincScheduleStart

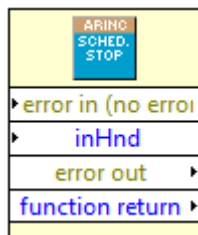


The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincScheduleStart().

```
EXPORT int CALLTYPE nativeArincScheduleStart(const uint32_t inHnd);
```

See page 49 for the ARINC 825 library function description.

## 8.5.14 ArincScheduleStop



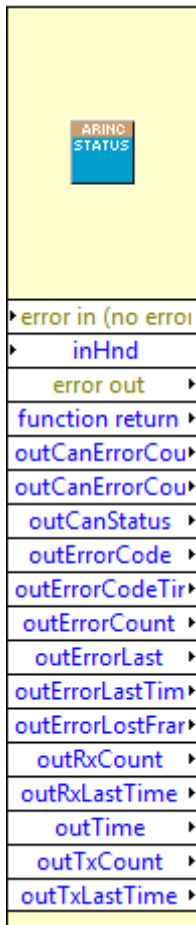
The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincScheduleStop().

```
EXPORT int CALLTYPE nativeArincScheduleStop(const uint32_t inHndProject);
```

See page 51 for the ARINC 825 library function description.



## 8.5.15 ArincStatus

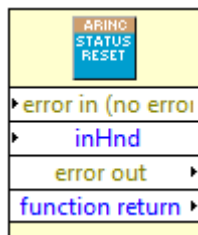


The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of `arincStatusReset()`.

```
EXPORT int CALLTYPE nativeArincStatusReset(const uint32_t inHnd);
```

See page 37 for the ARINC 825 library function description.

## 8.5.16 ArincStatusReset



The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of arincStatusReset().

```
EXPORT int CALLTYPE nativeArincStatusReset(const uint32_t inHnd);
```

See page 38 for the ARINC 825 library function description.

## 8.5.17 ArincTimeGet

ARINC TIME GET
▶ error in (no error)
▶ inHnd
error out ▶
function return ▶
outTime ▶
outTimeStatus ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincTimeGet().

```
EXPORT int CALLTYPE nativeArincTimeGet(const uint32_t inHnd, uint64_t * const outTime, uint32_t * const outTrigBStatus);
```

See page 41 for the ARINC 825 library function description.

## 8.5.18 ArincTxObjDisable

ARINC TX OBJ DISABLE
▶ error in (no error)
▶ inARINC_CMSG
▶ inFlag
▶ inHnd
error out ▶
function return ▶
inoutNum out ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincTxObjectDisable().

```
EXPORT int CALLTYPE nativeArincTxObjDisable(const uint32_t inHnd, ARINC_CMSG_T * const inCmsg, int32_t * const inoutNum, int32_t inFlag);
```

See page 54 for the ARINC 825 library function description.

## 8.5.19 ArincTxObjUpdate

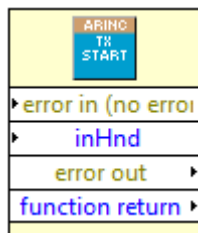
ARINC TX OBJ UPDATE
▶ error in (no error)
▶ inARINC_CMSG
▶ inHnd
error out ▶
function return ▶
inoutNum out ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of `arincTxObjUpdate()`.

```
EXPORT int CALLTYPE nativeArincTxObjUpdate(const uint32_t inHnd, ARINC_CMSG_T * const inoutCmsg, int32_t * const inoutNum);
```

See page 53 for the ARINC 825 library function description.

## 8.5.20 ArincTxStart

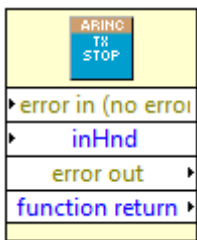


The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of arincTxStart().

```
EXPORT int CALLTYPE nativeArincTxStart(const uint32_t inHnd);
```

See page 46 for the ARINC 825 library function description.

## 8.5.21 ArincTxStop



The "native" VI set directly calls ARINC825 library functions. Please refer to the documentation of arincTxStop().

```
EXPORT int CALLTYPE nativeArincTxStop(const uint32_t inHnd);
```

See page 48 for the ARINC 825 library function description.

## 8.5.22 ArincWaitForTimeslot

ARINC WAIT TIME SLOT
▶ error in (no error)
▶ inHnd
▶ inTimeout
error out ▶
function return ▶

The "native" VI set directly calls ARINC825 library functions.  
Please refer to the documentation of arincWaitForTimeslot().

```
EXPORT int CALLTYPE nativeArincWaitForTimeslot(const uint32_t inHnd, const uint32_t inTimeout);
```

See page 55 for the ARINC 825 library function description.



## 9. Order Information

Type	Properties	Order No.
<b>Software</b>		
ARINC825-LCD Windows/Linux	ARINC 825 object licence for Windows® and Linux® CD+Licence Usable with all esd ACC based CAN interfaces - object licence for Windows/Linux - ARINC 825 dll's/lib's - Lab VIEW VI-Set for ARINC 825 - documentation	C.1140.06
ARINC825-LCD RTX CD+Licence	ARINC825-LCD RTX/RTX64 CD+Licence Usable with all esd ACC based CAN interfaces - object licence for RTX/RTX64 - ARINC 825 dll's/lib's - documentation	C.1140.16
ARINC825-LCD QNX CD+Licence	ARINC825-LCD QNX® CD+Licence Usable with all esd ACC based CAN interfaces - object licence for QNX - ARINC 825 dll's/lib's - documentation	C.1140.17
ARINC825-LCD VxWorks CD+Licence	ARINC825-LCD VxWorks® CD+Licence Usable with all esd ACC based CAN interfaces - object licence for VxWorks - ARINC 825 dll's/lib's - documentation	C.1140.18

For detailed information about the driver availability of your special operating system, please contact our sales team.

**Table 1:** Order information

### PDF Manuals

Manuals are available in English and usually in German as well. For availability of manuals see table below. Please download the manuals as PDF documents from our esd website [www.esd.eu](http://www.esd.eu) for free.

Manuals	Order No.
ARINC 825 Library-ME	Software manual in English C.1140.21

**Table 2:** Available manuals

### Printed Manuals

If you need a printout of the manual additionally, please contact our sales team: [sales@esd.eu](mailto:sales@esd.eu) for a quotation. Printed manuals may be ordered for a fee.