SIEMENS

ICs for Communications

HDLC / ASYNC Converter

Based on SAB 82532 and SAB 80C166

Application Note 08.93

SAB 82532 Revision History:	Original Version 08.93
Previous Releases:	
Page	Subjects (changes since last revision)

Data Classification

Maximum Ratings

Maximum ratings are absolute ratings; exceeding only one of these values may cause irreversible damage to the integrated circuit.

Characteristics

The listed characteristics are ensured over the operating range of the integrated circuit. Typical characteristics specify mean values expected over the production spread. If not otherwise specified, typical characteristics apply at T_A = 25 °C and the given supply voltage.

Operating Range

In the operating range the functions given in the circuit description are fulfilled.

For detailed technical information about **“Processing Guidelines”** and **“Quality Assurance”** for ICs, see our **“Product Ovewrview”**.

Edition 08.93
This edition was realized using the software system FrameMaker®.

**Published by Siemens AG, Bereich Halbleiter, Marketing-Kommunikation,
Balanstraße 73, 81541 München.**

Packing

Please use the recycling operators known to you. We can also help you - get in touch with your nearest sales office. By agreement we will take packing material back, if it is sorted. You must bear the costs of transport.

For packing material that is returned to us unsorted or which we are not obliged to accept, we shall have to invoice you for any costs incurred.

Table of Contents **Page**

SAB 82532 with SAB 80C166

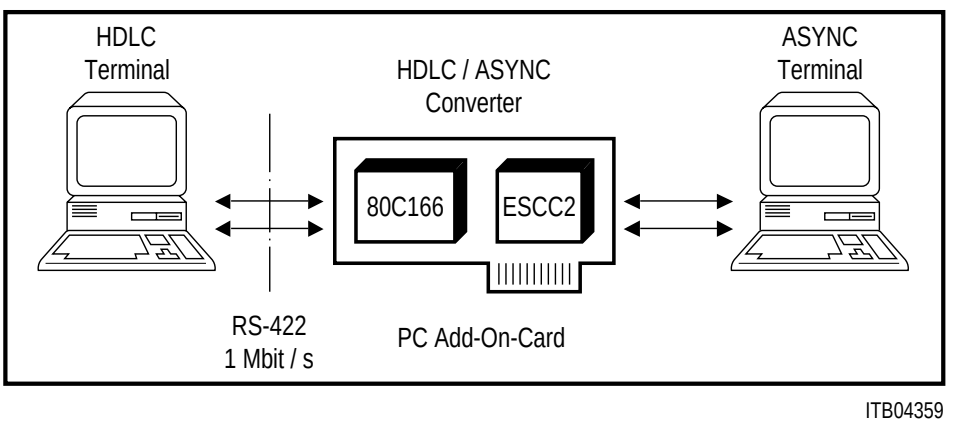
HDLC/ASYNC Converter

Summary

This application note describes a simple application example for an HDLC/ASYNC Converter using the ESCC2 (SAB 82532) and the 16-Bit-Microcontroller SAB 80C166. Transfer rates up to 1 Mbit/s are supported. This application note is based on a real world and fully tested communication subsystem. It demonstrates the advantages of using the ESCC2 in combination with the SAB 80C166 for general communication purposes.

Contents

- Chapter 1
  General Overview
- Chapter 2
  How to integrate the ESCC2 into an 80C166 Microprocessor System. All hardware aspects concerning the ESCC2 and 80C166 are explained.
- Chapter 3
  Description of the different software levels and their realization. A brief introduction to the basic operating software and the corresponding Application Program Interface (API) for the C-language is provided.
  A minimum set of only 13 interface functions and macros is necessary to integrate the device driver and conversion software.



Functional Block Diagram

- Chapter 4
  Detailed description of how the ESCC2 can be programmed to handle HDLC- and ASYNC-communication for higher data rates. The basic architecture of an object oriented Device Driver Module for the ESCC2 is explained. Further, it is explained how user requests for frame transmission and device interrupts can be handled concurrently in the most effective way.

The way how to serve multiple requests and frame lengths greater than 32 bytes is described in form of finite state machines and corresponding flow diagrams.

- Chapter 5
  A very simple application program module for the frame conversion on top of the ESCC2-device driver software and the underlying run-time software is described.
- Appendix A
  Detailed schematics for the hardware.
- Appendix B
  Corresponding PAL-equations (1 PAL).
- Appendix C
  Source code listings of all relevant software modules.
- Appendix D
  A make file is included, which allows to build the software using the software development tools from Tasking.

Remarks

It is recommended to have a basic understanding of the ESCC2-interrupt interface and the 2×32 byte FIFO-structure described in the ESCC2 Technical Manual. For details concerning the SAB 80C166 please refer to the corresponding User's Manual.

For this application note the 80C166-compiler and assembler from Boston Systems Office/Tasking have been used. The software is completely written in C, with the exception of file DDSHSERV.SRC, an assembler file containing the processor specific interrupt interface (setting interrupt frames and vectors).

This application note has been realized with a PC-add-on board to shorten the software development cycles. It supports a fast download of device driver software onto the communication subsystem directly from the PC. The application example itself has been designed as a stand-alone solution to reduce the requirements on an individual environment. This shall give you a first impresssion of how easy it is to combine the ESCC2 with the 80C166 for communication subsystems in general. Therefore the complexity of the software environment has been reduced to a minimum.

The data transfer rate of 1 Mbit/s has been chosen for illustration purpose only. A higher data transfer rate is possible.

1 Introduction

This application note describes a design example for an HDLC/ASYNC converter using the SAB 82532 (Enhanced Serial Communication Controller – 2 Channels, ESCC2) and the microcontroller SAB 80C166. It is recommended that before reading this application note, the user has a basic understanding of the interrupt interface and FIFO-structure of the SAB 82532, described in the “ESCC2 Technical Manual”.

In this description of the HDLC/ASYNC converter two basic subjects are discussed. The first one is how the SAB 82532 can be integrated into a hardware environment based on a 16-bit microcontroller from Siemens, the SAB 80C166. One approach is to treat the SAB 82532 as a peripheral device connected to the external bus of the SAB 80C166. In this case practically no glue logic in the form of external hardware is required.

The second subject is the software of the HDLC/ASYNC converter. The special device driver for the ESCC2 has the advantage of utilizing the SAB 82532 features concerning different transfer modes (HDLC, ASYNC), interrupt treatment and FIFO-structure to enhance performance.

Due to their 16-bit system interface and all other inherent powerful features, the ESCC2 and the 80C166 build a strong combination well suited for high speed communication and protocol conversion in general.

This application example has been written in C using the C-Compiler and Assembler 80C166 from Boston Systems Office/Tasking.

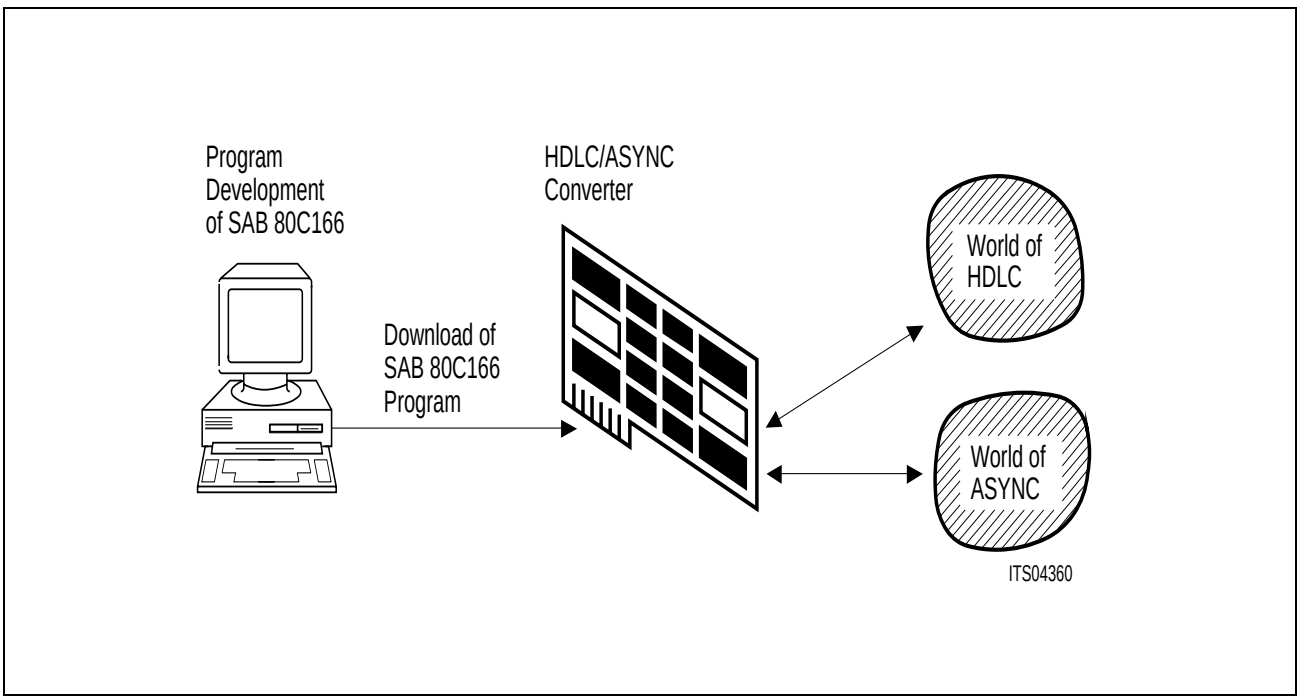


Figure 1
Application HDLC/ASYNC Converter

2 Hardware of the HDLC/ASYNC Converter

The hardware of the HDLC/ASYNC converter is realized as a plug-in PC-board. Beside the microcontroller SAB 80C166 the following components are used: a 64 K EPROM containing the firmware, a 256 K RAM and a PAL for address decoding.

Figure 2 shows the functional block diagram of the hardware. A detailed schematic can be found in Appendix A.

The Memory Organization

The SAB 80C166 provides a total addressable memory space of 256 Kbytes. This address space is arranged in four segments of 64 Kbytes each, and each segment is again subdivided in four pages of 16 Kbytes each. **Figure 3** gives an overview of the memory organization of the HDLC/ASYNC converter. The equations to decode the corresponding chip select signals are listed in Appendix B.

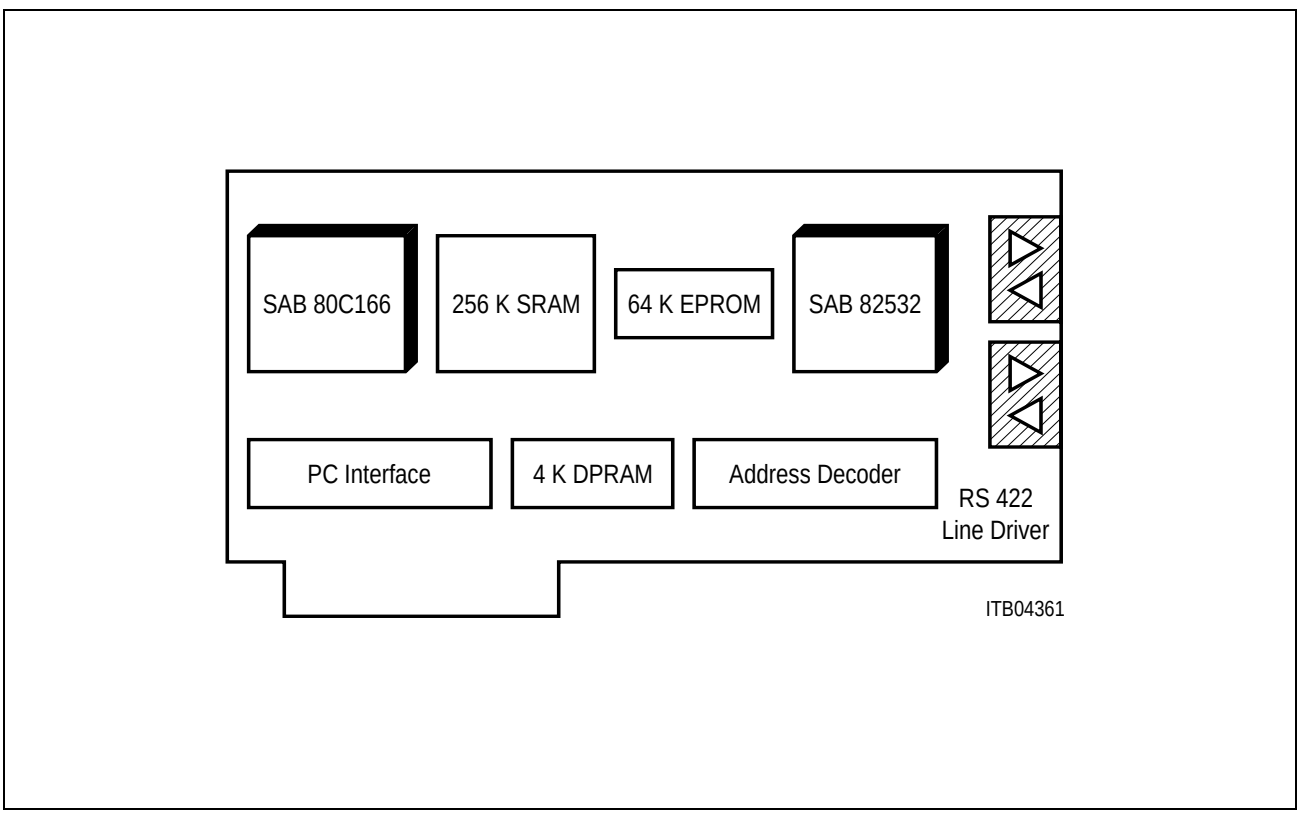


Figure 2
Functional Block Diagram

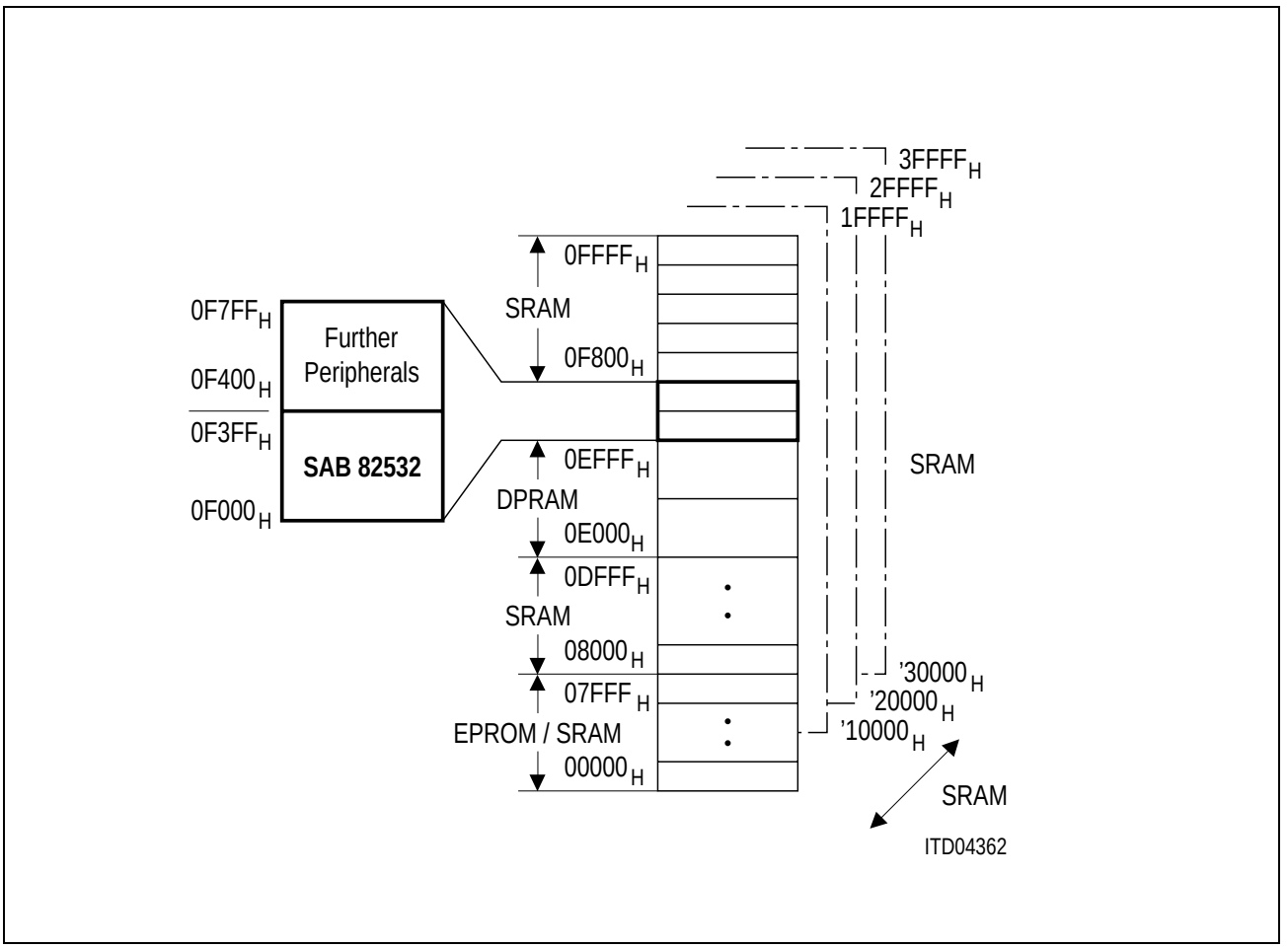


Figure 3
Memory Organization

Microprocessor Interface

Practically no external components are required to adapt the SAB 82532 to the SAB 80C166. The SAB 82532 is directly connected to the data bus and address bus of the SAB 80C166 system. The $\overline{RD}$-, $\overline{WR}$- and $\overline{BHE}$-signals are also connected directly.

The $\overline{CS}$ for the SAB 82532 is generated in a PAL. WIDTH is connected to + 5 V, which configures the ESCC2 into 16-bit bus mode.

Interrupt

Since no interrupt acknowledge cycle is supported by SAB 80C166 pin $\overline{INTA}$ is deactivated by connecting $\overline{INTA}$ to + 5 V. Because of the single device application IE0 and IE1 are not used; they are tied to GND. The INT pin is connected to P2.1 (CC1IO); it has to be considered that the interrupt is (positive) edge triggered. To avoid loosing an interrupt it is recommended masking interrupts for a short time before leaving the interrupt function. In this way a new edge can be forced, because the SAB 82532 safeguards the interrupts even while being masked.

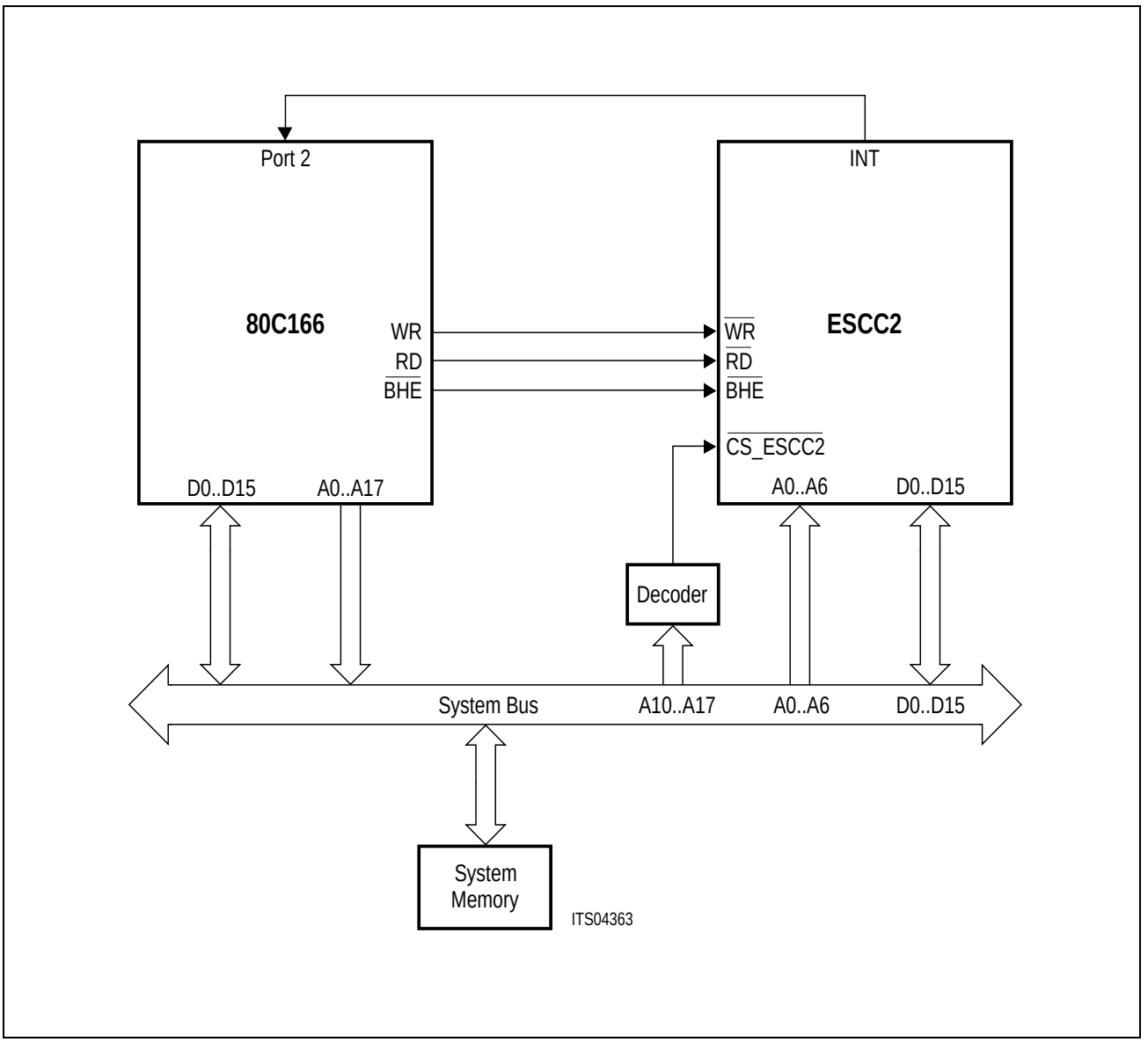


Figure 4
Architecture of System

DMA Interface

$\overline{\text{DACKA}}$ and $\overline{\text{DACKB}}$ are tied to + 5 V, since no DMA-transfer is supported.

Reset

It should be considered that RES is an active high input signal, so that the $\overline{\text{RSTOUT}}$-signal of the SAB 80C166 has to be inverted.

Serial Interface

The input signals RxDn, $\overline{\text{CTS}}$n, CDn, RxCLKn (n = A, B) are pulled up to + 5 V by a resistor of 10 kΩ. Modem control functions are not supported in this application, therefore $\overline{\text{RTS}}$n and $\overline{\text{CTS}}$n are connected directly.

The physical line is realized with a RS-422 compatible line driver interface, consisting of the driver circuit AM26LS30 and the receiver circuit AM26LS32. These circuits are connected to the TxDn and RxDn signals of the SAB 82532.

The Timing Characteristics

The timing of the microcontroller can be adapted to that of the ESCC2 by changing by software the default values of the SAB 80C166 (refer to SAB 80C166 Technical Manual).

Starting with the address setup time t_{SU} (A), which is specified as 0 ns for the microcontroller and > 5 ns for the SAB 82532, a read/write delay has to be introduced. With the delay programmed, the falling edge of the ALE-signal leads the falling edges of the $\overline{\text{RD}}$ or $\overline{\text{WR}}$ by a quarter of a machine cycle (25 ns at f_{OSC} = 40 MHz). This delay does not extend the memory cycle time, and thus it does not slow down the controller in general.

Because of the read/write delay discussed above the $\overline{\text{RD}}$-$\overline{\text{WR}}$-pulse width is shortened from 65 ns to 40 ns. The SAB 82532 specification however requires a $\overline{\text{RD}}$-pulse width of > 60 ns; therefore one wait state has to be programmed. One memory cycle time wait state requires half a machine cycle (50 ns at f_{OSC} = 40 MHz).

The address hold time t_H(A) of the SAB 82532 has to be equal or longer than 10 ns (SAB 80C166: 0 ns). With an additional memory tristate time wait state the hold time has to be lengthened.

The timing can be modified by programming the bus configuration register or by modifying the start-up code of the system. An advantage is that the timing can be modified individually in a selected address space.

3 Device Driver System

This chapter gives an overview about the system software used to implement the application specific software modules. This Device Driver System (DSS) provides a simple fundamental platform for the integration of Device Driver Modules (DDMs) and Application Program Modules (APMs).

Because of its reduced complexity and its high degree of portability it is possible to concentrate on the main issues related to writing device driver code for the Enhanced Serial Communication Controller (ESCC2) from Siemens.

3.1 Overview

The basic concept of the device driver system relies on the transfer of messages between DDMs and APMs. A DDM contains the low level device driver functions, including the interrupt service routines, which must fulfill individual real-time constraints. As opposed to the DDM an APM combines the more intelligent time consuming data (message) processing functions. The general concept is shown in **figure 5**.

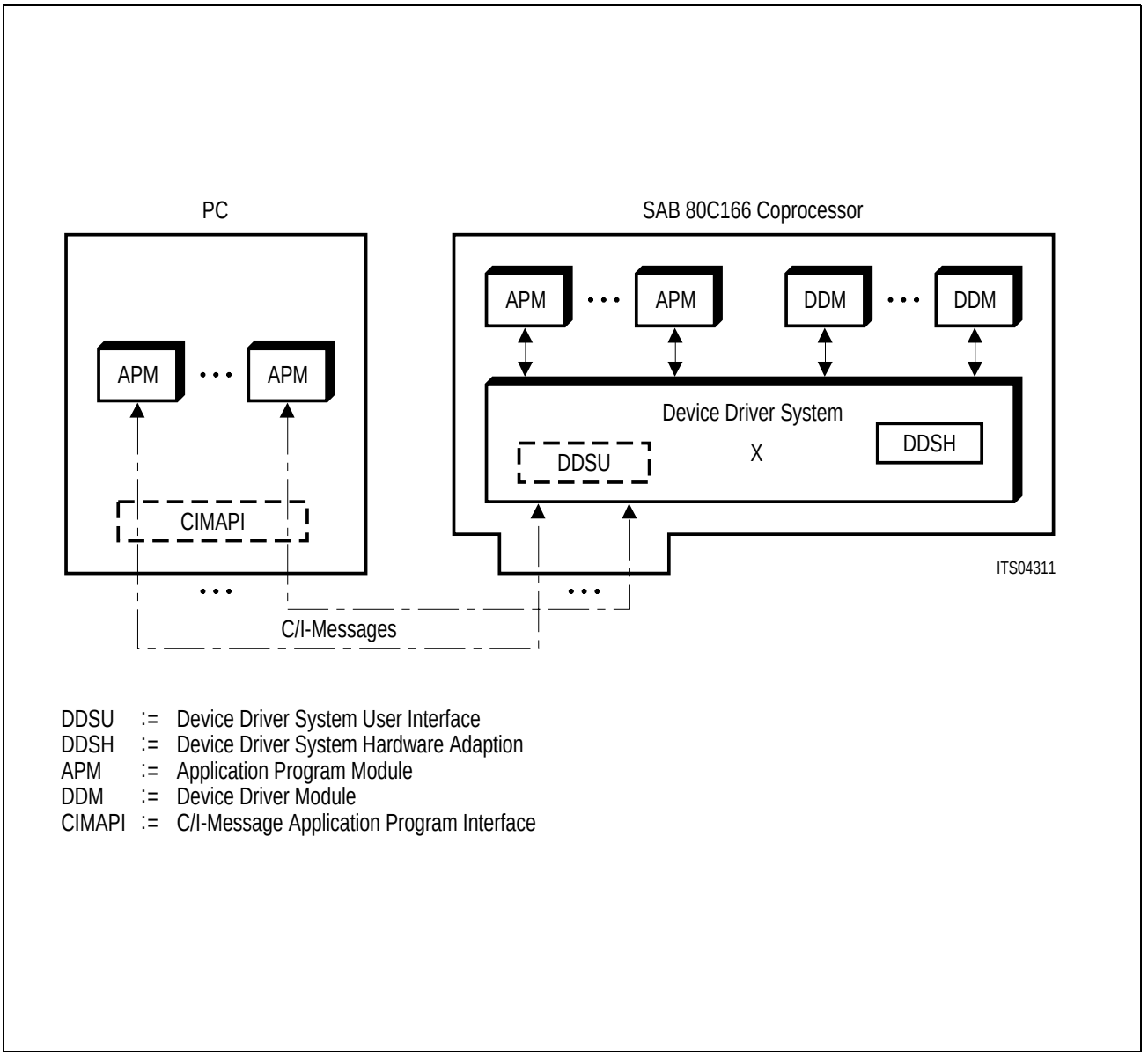


Figure 5
Device Driver System Concept

The device driver system maintains a unique data structure, the Command/Indication Message (C/I-Message) to transfer information between the different APMs and DDMs. The standard format is depicted in **figure 6**.

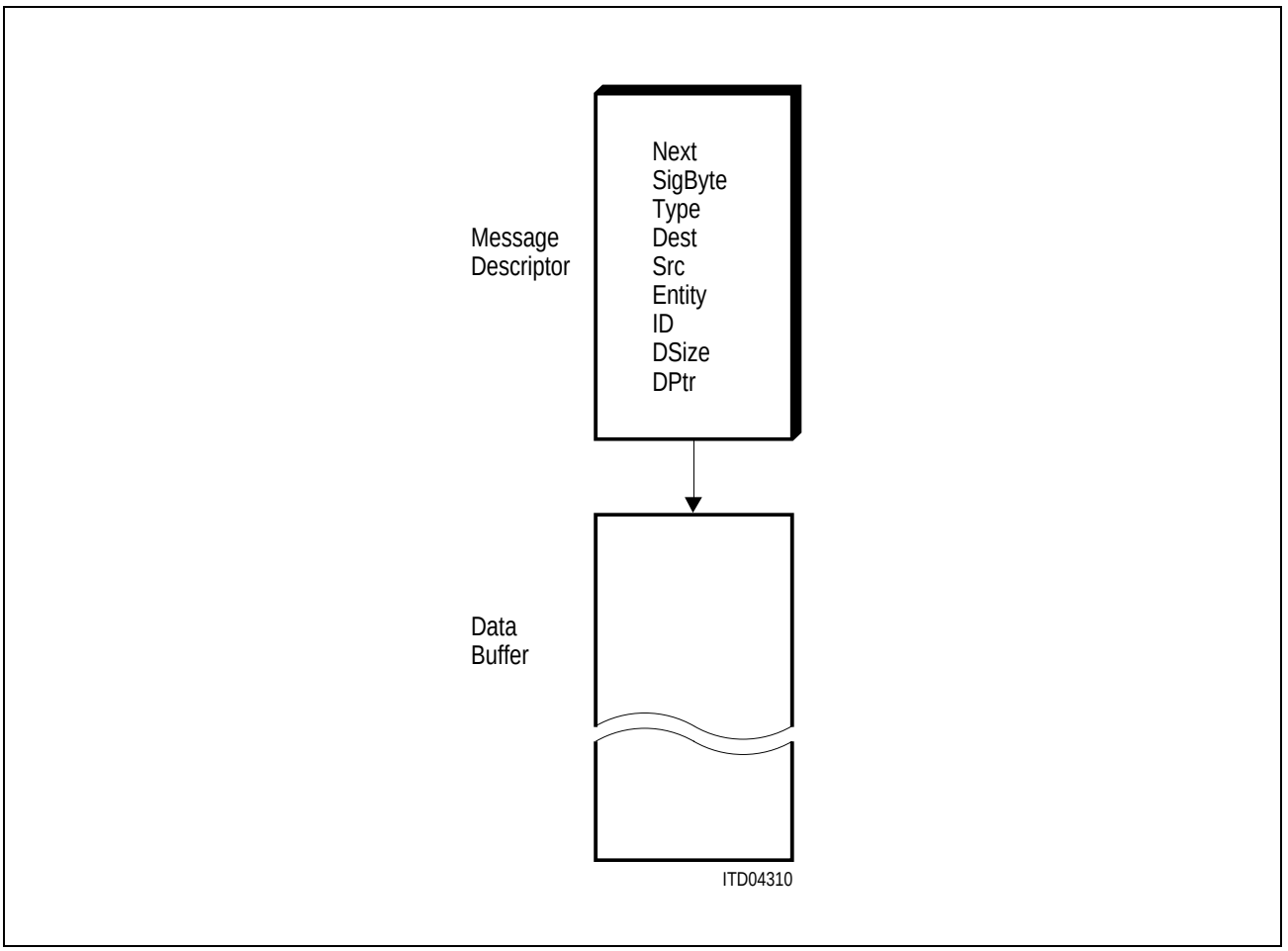


Figure 6
C/I-Message Format

The device driver system itself provides all services necessary to build a high performance communication system. The main tasks are summarized below:

- Initial Hardware Initialization
- Initial Software Initialization
- Main Program Control
- Message Routing
- Message Buffer Management
- Timer Management
- System Control
- User Interface

3.2 General Module Architecture

A module to be integrated into the device driver system's environment must fulfill only a minimum set of requirements. **Figure 7** illustrates the basic architecture of a device driver module.

First of all a module, a Device Driver Module (DDM) or an Application Program Module (APM), must have one general function to be used as message entry point. In addition to that, a DDM has to assign and initialize its interrupt entry point and the corresponding modes. When the whole system

is started, the device driver system sends an initial C/I-message (INIT_MODULE) to every module. This gives every module the opportunity to establish its own context, initialize its data structures and if necessary the related devices. Following this initialization an integrated module can receive C/I-messages, which are decoded inside the module. Additional service functions, described in section 3.5, are needed inside a module (e.g.: to request and send C/I-messages, en-/disable interrupts, …).

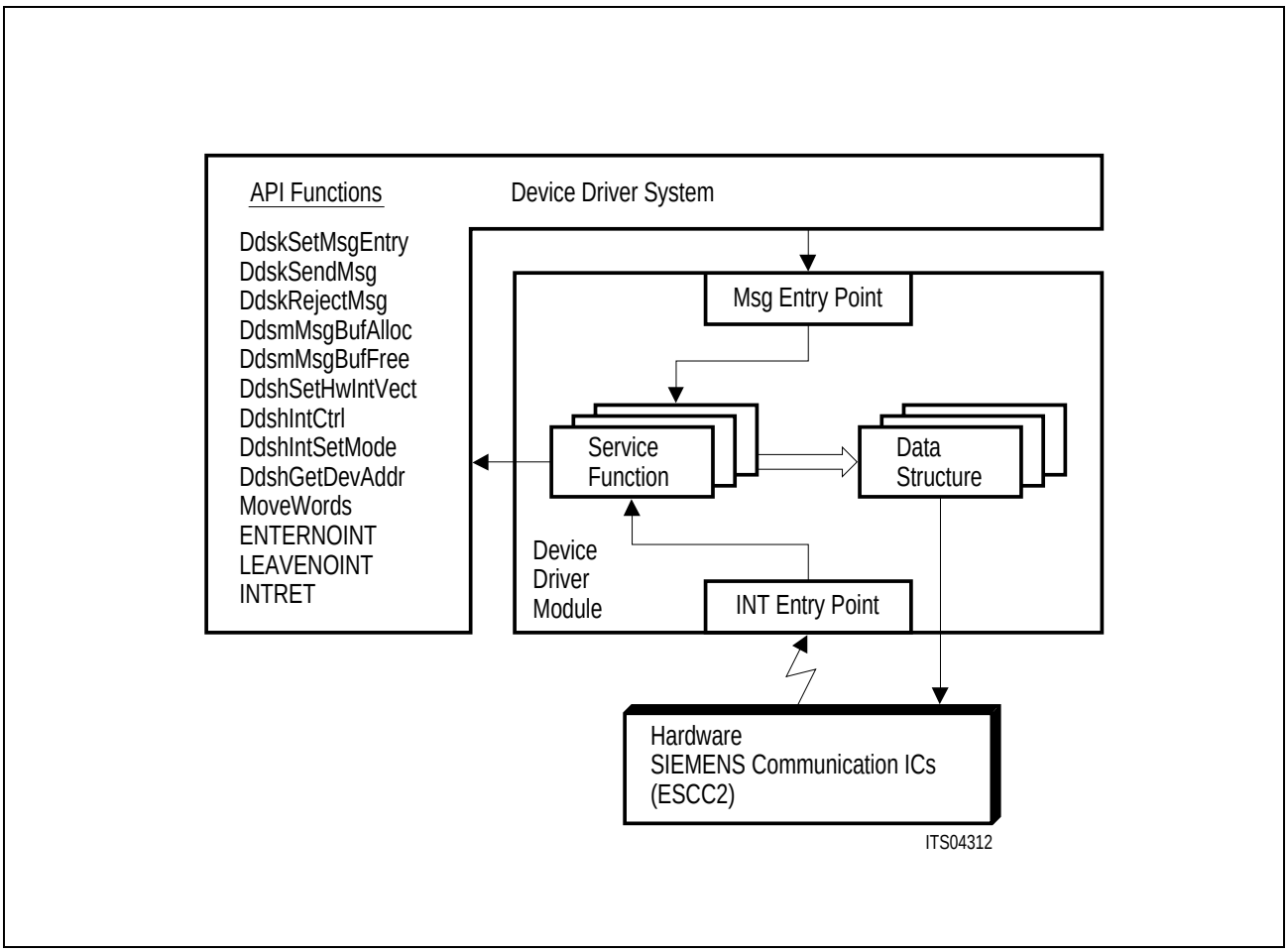


Figure 7
General Module Architecture

3.3 Integration of Modules

To integrate individual modules into the device driver system's environment one of the modules must have the unique function **DdsIntegrate**. This generic function is called during the system start-up phase. Its purpose is to assign the message entry points for the application program and device driver modules by means of the service function **DdskSetMsgEntry**.

3.4 The Example "HDLC-ASYNC Converter"

The application example described in this document has been implemented on a PC-coprocessor board using the new 16-bit microcontroller 80C166 from Siemens.

A functional block diagram of the complete example is shown in **figure 8**.

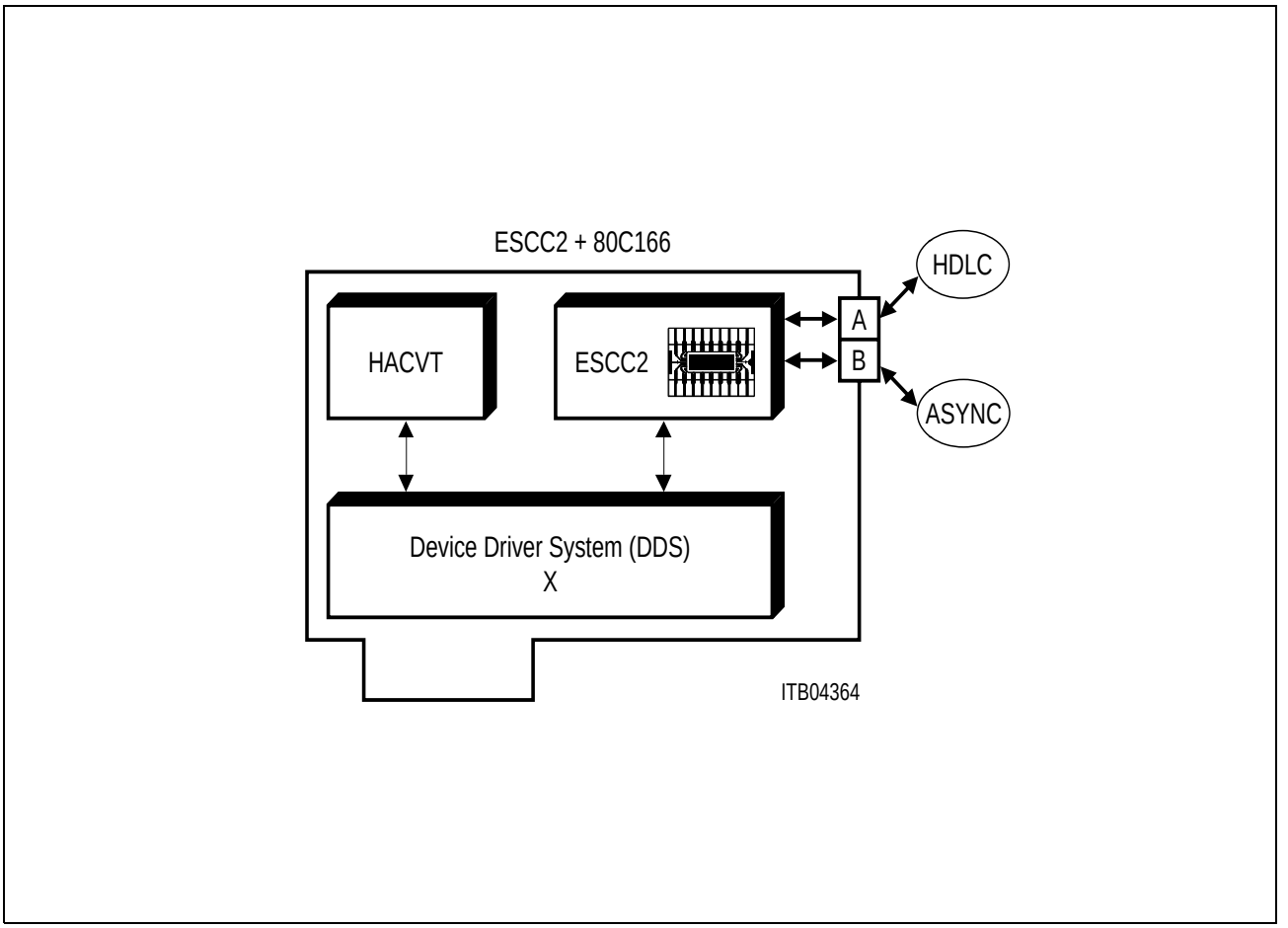


Figure 8
Structure of Converter Software

To concentrate on the device driver software and the application of the ESCC2 in a communication system, a minimum interface to the device driver system and the hardware specific adaptions (see section 3.5 and Appendix C) are described in more detail.

Because the PC-user interface is highly specialized and used only for the downloading of the run-time software onto the PC-board it is not described here. No timer functions are required in this application, so they are omitted as well.

The following **figure 9** shows the information flow inside the whole communication subsystem, especially how the different software layers (modules) are involved. The conversion from HDLC to ASYNC is illustrated, assuming an incoming HDLC-frame length greater than 32 bytes. The ESCC2-device driver module handles the receive interrupts (see chapter 4; RPF and RME), and translates the complete received HDLC-frame into a corresponding C/I-message. All C/I-messages are inserted into a single message queue, maintained by the device driver system's kernel. The kernel routes all messages to the appropriate destination module. The C/I-message RC_HDLC_FRAME_OK for instance, related to an incoming HDLC-frame via ESCC2-channel A, is routed to the HACVT-module. The HACVT-module reacts by converting the HDLC to an ASYNC-frame and sending a SEND_FRAME message to the ESCC2-device driver module.

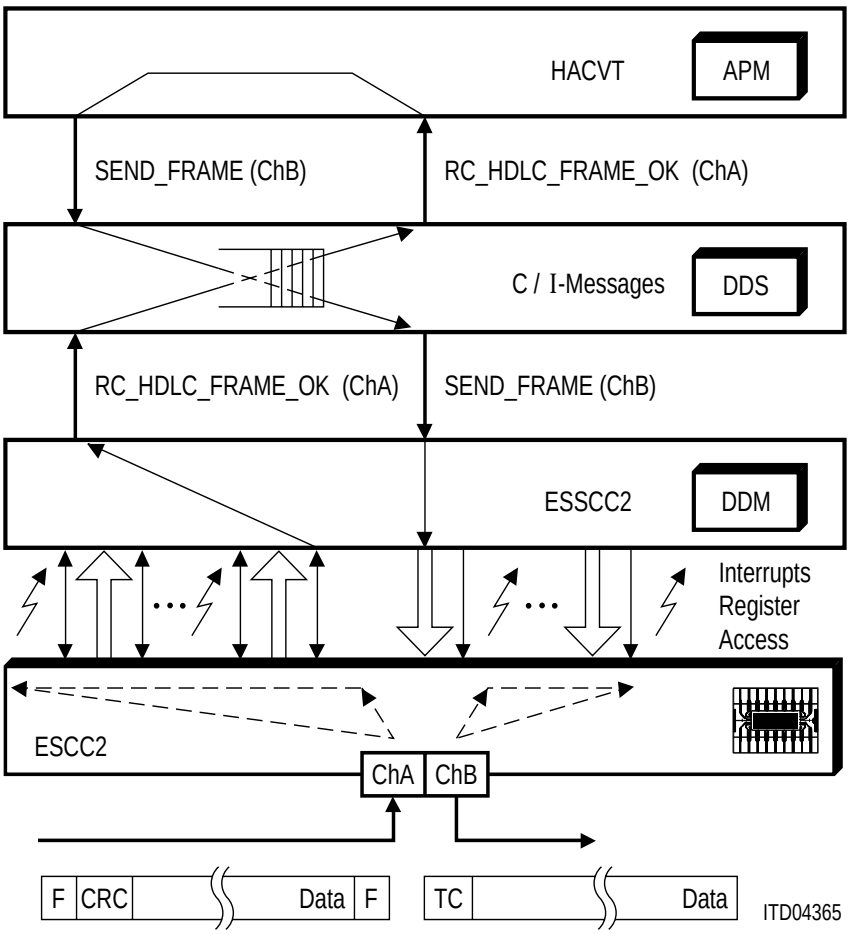


Figure 9
Information Flow HDLC to ASYNC

One element of the C/I-message, the parameter entity (**see figure 8**), determines the ESCC2-channel to be used for transmission and the source for a received frame, be it an HDLC-frame via channel A (entity = 0) or an ASYNC-frame via channel B (entity = 1). In this way the HACVT-module converts an incoming HDLC-frame (entity = 0) to an outgoing ASYNC-frame (entity = 1), by modifying the C/I-message descriptor appropriately.

ciMsg → Dest = ESCC2_MODULE;
ciMsg → Src = HACVT_MODULE;
ciMsg → Entity = CHANNEL_B;
ciMsg → ID = SEND_FRAME;

The data buffer of the converted C/I-message remains the same (DPtr; for more details refer to chapter 5).

The SEND_FRAME message will force the ESCC2-module to transmit the data buffer content either as an HDLC-frame or an ASYNC-frame depending on the parameter entity. It is again the responsibility of the ESCC2-module to translate the send command into appropriate interactions with the ESCC2-device and to serve the corresponding transmit interrupts (see chapter 4; XPR).

This approach of using a separate Device Driver Module (DDM) for real-time functions and a corresponding Application Program Module (APM) as a specific message processor is a very powerful concept, because the probability of loosing interrupt events is reduced to a minimum.

3.5 Application Program Interface

The following description briefly explains the main purpose of the device driver system's service functions and how they are called in C. No distinction between functions realized in C and in assembly code is made on this level. The corresponding function prototypes, macro definitions, typedefs and defines are included either in the header file DDSG.H or DDSH.H.

All hardware specific functions and macros belonging to the individual processor system (e.g.: processor type, memory layout, …) are combined in one system module DDSH, which is available in source code (see Appendix C). To transfer the device driver source code to another processor system the main work consists in modifying the functions and macros in the DDSH-module so that they fit into the new system.

Moreover you will find definitions concerning the C/I-messages in the MSG.H file, like module and message IDs and related parameters (see also Appendix C).

DdskSetMsgEntry (int *destId*, (*msgEntryFctPtr)

Writes the pointer to the module entry function *msgEntryFctPtr* in the device driver system's message routing table in dependance on the module destination identifier destId.

DdskSendMsg (CIM_MSG_DESCR_PTR *ciMsg*);

Puts a C/I-message *ciMsg* into the device driver system's central message queue. A module which wants to send a message to another module calls this function with a pointer to the previously prepared message buffer. This message buffer can be requested from the buffer management by means of the function DdsmMsgBufAlloc (see below). As a minimum the C/I-message must contain the destination ID of the addressed module, its own source ID and the message ID itself. In addition to that, the data buffer must be filled with the appropriate information, if any. The C/I-message format and the corresponding structure CIM_MSG_DESCR_PTR are specified in the source file DDSG.H (see Appendix C).

DdskRejectMsg (CIM_MSG_DESCR_PTR *ciMsg*, int *reason*);

In case a just received C/I-message *ciMsg* contains wrong information, it may be rejected by means of this function. The value *reason* may be used to signal why this message has been rejected.

ciMsg = **DdsmMsgBufAlloc** (int size);
CIM_MSG_DESCR_PTR *ciMsg*;

Returns a pointer *ciMsg* to the next available C/I-message buffer with a data buffer assigned to it, depending on the parameter size. When there is no message buffer available, a NULL pointer will be returned instead.

DdsmMsgBufFree (CIM_MSG_DESCR_PTR *ciMsg*);

Releases a previously allocated C/I-message buffer referenced by the pointer *ciMsg*.

DdshSetHWIntVect (HW_INT_TYPE *intx*, INT_FCT_PTR *intFct*);

Installs an interrupt frame function, which calls the specified interrupt function *intFct*. The interrupt frame is initialized in the system module DDSH, which is dependant of the processor type (e.g.: 80C166).

DdshIntCtrl (HW_INT_TYPE *intx*, BOOL *enable*);

Enables or disables an individual interrupt source. Only device no. 1 is supported in this application.

DdshIntSetMode (HW_INT_TYPE *intx*, INT_MODE_TYPE *mode*);

Sets interrupt mode. In this application only device no. 1 and mode 1 is supported. Mode 1 means for the 80C166 that the interrupt is triggered on positive external transitions on pin CC1IO.

devAddr = **DdshGetDevAddr** (DEVICE_TYPE *dev*);
WORD32 *devAddr*;

Returns the base address *devAddr* of the specified device *dev*. Only device no. 1 is supported in this application.

MoveWords (W16PTR s*rcPtr*, W16PTR *destPtr*, WORD *byteCnt*);

Optimized procedure for a fast transfer of a memory block with the specified length *byteCnt* from source address *srcPtr* to the destination address *destPtr*.

ENTERNOINT (WORD *cpuState*);

Is a special macro defined in DDSH. Disables temporarily all interrupts. The current CPU-state is stored in *cpuState*. This allows nesting of interrupts in conjunction with macro LEAVENOINT.

LEAVENOINT (WORD *cpuState*);

Is a special macro defined in DDSH. Restores the old CPU-state by means of the variable *cpuState*. It puts a previously interrupted program state back the way it was before ENTERNOINT was called. This allows nesting of interrupts in conjunction with macro ENTERNOINT.

INTRET

Is a special macro defined in DDSH. It releases the interrupt logic of the corresponding interrupt line. This is necessary to allow the interrupt controller to accept interrupts. Should be called whenever an interrupt routine is left.

4 The ESCC2-Device Driver Module

4.1 Overview

The device driver module ESCC2, integrated into the device driver system's environment, represents the connection to the Siemens data communication device ESCC2 (Enhanced Serial Communication Controller, ESCC2). Therefore, referring to the programming of the device by the user, this module has to be understood as the interface module between the applications level and the device itself. The functional description of the ESCC2 device driver module is shown in **figure 10**.

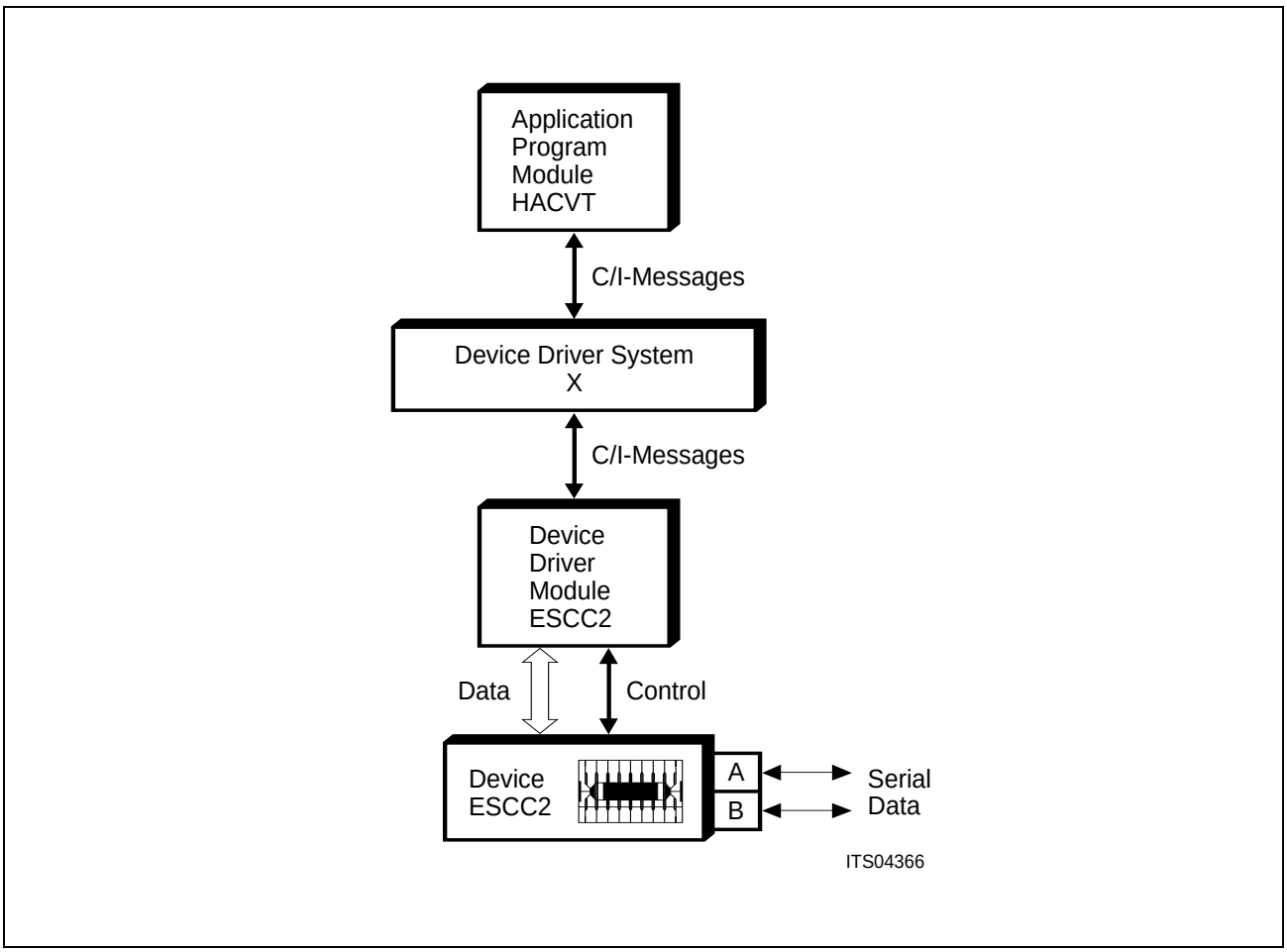


Figure 10
The ESCC2-Device Driver Module

Supported by the device driver system the application program module HACVT is able to direct individual tasks to the ESCC2-device. The ESCC2-device driver module receives these messages and passes them to the ESCC2, matching the device's programming specification. Commands and/or data may be written to the ESCC2. Status and control information is read from the appropriate ESCC2-registers.

In addition, the device driver module serves the interrupts generated by the ESCC2-device itself. In this case a message to the application module can be sent.

The source code of the driver module is found in Appendix C. The module ESCC2.H contains the defined values, data structures and function types, whereas all C-functions concerning ESCC2 are to be found in ESCC2.C.

4.2 Basic Data Structure of the ESCC2-Device Driver Module

Since the ESCC2-device has two symmetrical channels operating independently, two data links can be supported. These data links are controlled by a number of parameters contained in two structures of the type DATA_LINK_CTRL (see ESCC2.H), one for each channel. These are the fundamental data structures the ESCC2-device driver module is based on.

Detailed description of the elements:

ESCC2_REG_MAP_ptr Escc2: The first problem when programming the ESCC2 is the access to its registers. For the sake of clarity the registers are gathered in structures of registers, depending on the mode (HDLC, ASYNC or BISYNC) and depending on the kind of access (READ or WRITE). The offset of a single register in this structure corresponds to the register address (A0… A6).

```
typedef struct
{
        WORD  fifo [16];  /*  RFIFO                 */
        BYTE    star;     /*  Status Register       */
        BYTE    rsta;     /*  Receive Status Reg.   */
        BYTE    mode;     /*  Mode Register         */
        BYTE    timr;     /*  Timer Register        */
        BYTE    xad1;     /*  Transmit Address      */
.
.
.
        BYTE    gis;      /*  Global Int. Status    */
        BYTE    ipc;      /*  Interrupt Port Conf.  */
        BYTE    isr0;     /*  Interrupt Status 0    */
        BYTE    isr1;     /*  Interrupt Status 1    */
        BYTE    pvr;      /*  Port Value Register   */
        BYTE    pis;      /*  Port Interrupt Status */
        BYTE    pcr;      /*  Port Conf. Reg.       */
        BYTE    not_used_9;
}
HDLC_MODE_READ;
```

Since the registers in the different modes and for different kinds of access share the same address region, these structures are combined in a union type definition, called ESCC2_REG_MAP.

```
typedef union
{
        HDLC_MODE_READ      Hdlc_Rd;
        HDLC_MODE_WRITE     Hdlc_Wr;
        ASYNC_MODE_READ     Async_Rd;
        ASYNC_MODE_WRITE    Async_Wr;
        BISYNC_MODE_READ    Bisync_Rd;
        BISYNC_MODE_WRITE   Bisync_Wr;
}
ESCC2_REG_MAP;
```

Now, that the relative location (= offsets) of the single registers are known for different modes and kinds of access, the base address of this total register map can be defined. Seeing that the ESCC2 is a data communication device with two symmetrical serial channels using different memory regions each control structure has its own base address. These addresses are defined during the Escc2Init () routine. One gets the base address of the device in memory by executing DdshGetDeviceAddress (*device*).

```
ESCC2_REG_MAP_ptr           base;
.
.
base = DdshGetDeviceAddress (DEVICE1);
DataLinkCtrl [ESCC2_CH_A].ESCC2 = base;
base        =  (HSCX_REG_MAP far*)
               (((DWORD) base) + 0x40);
DataLinkCtrl [ESCC2_CH_B].ESCC2 = base;
```

int ChID: The channel identification *ChID* conforms to the index of the actual data link control structure (DataLinkCtrl[index].ChID = index). Like the Source the ChID is assigned during the initialization of the corresponding ESCC2-channel:

```
.
.
.
Dlc              = & DataLinkCtrl[command → Entity];
Dlc → ChID       = command → Entity;
Dlc → Source     = command → Src;
.
.
```

int Mode: Each channel operates in its own transfer mode: HDLC, ASYNC etc.. *Mode* contains this transfer mode.

W16PTR FIFOfe: This pointer to the Tx/Rx-FIFO of the corresponding channel allows a fast access during the execution of an interrupt routine.

W16PTR IsrReg: This pointer to the interrupt status register of the corresponding channel allows a fast access during the execution of an interrupt routine.

W16PTR ImrReg: This pointer to the interrupt mask register of the corresponding channel allows a fast access during the execution of an interrupt routine.

W16PTR CmdReg: This pointer to the command register of the corresponding channel allows a fast access to the command register during the execution of an interrupt routine.

WORD16 IntMask: This element contains the channel and/or mode specific interrupt mask.

WORD8 TxShort: This element contains the channel and/or mode specific transmit command for short frames; because of this element it is possible to design transmitter service routines independent on mode (HDLC, ASYNC…).

WORD8 TxLong: This element contains the channel and/or mode specific transmit command for long frames; because of this element it is possible to design transmitter service routines independent on mode (HDLC, ASYNC…).

void (*ServeInterrupt) (void): This function pointer to the channel and/or mode specific interrupt service routine allows designing the treatment of the interrupt function independent on mode (HDLC, ASYNC…).

WORD8 Source: Since there may be different sources requesting a data link performed by the ESCC2 the data link control structure contains an element called *Source*. Keeping in mind the actual *Source*, the ESCC2-module is able to send the messages to the proper destination (e.g. protocol modules, LAPD or other application modules, HACVT).

```
.
.
Dlc → message → Entity    =  Dlc → ChID;
Dlc → message → Dest      =  Dlc → Source;
Dlc → message → Src       =  ESCC2_MODULE;
…
DdskSendMsg (Dlc → message);
.
.
```

CIM_MSG_DESCR_PTR Msg: Assuming that an ESCC2-channel receives a frame, a message buffer has to be allocated for this channel; *Msg* is the pointer to the channel's current receive message buffer.

```
.
.
Dlc → Msg = DdsmMsgBufAlloc (maxSize);
if (!Dlc → Msg)
{
Dlc → ESCC2 → Hdlc_Wr.cmdr = CMDR_RHR;
return;
}
.
.
```

CIM_MSG_DESCR_PTR head: Both channels use linked lists to store frames for transmission. These linked lists are first-in first-out queues, frames are added at the tail end and taken away from the head end. So head points to the item, which will be taken away as the next one. Tail points to the item which was added as the last one. The tail is controlled by sending frames to the ESCC2-channel; head is controlled by the channel specific transmitter interrupts.

CIM_MSG_DESCR_PTR tail: See CIM_MSG_DESCR_PTR *head*.

int RcState: The serial interface of the ESCC2 consists of two full duplex channels. So both channels are able to receive and transmit data at the same time. Normally a receiver or a transmit procedure consists of more than one interrupt routine; in this case various states are accepted. The current states are stored in *RcState* and *TxState*.

int TxState: See int *RcState*.

W8PTR RcCurr: The frames received (transmitted) by the ESCC2-channel may be longer than one FIFO-length. Then the FIFO has to be read (written) more than once and the receiver (transmitter) has to keep in mind the current position of the data stream in the allocated buffer. W8PTR *RcCurr* (W8PTR *TxCurr*) points to the current position of the data stream.

W8PTR TxCurr: See W8PTR *RcCurr*.

int TxCnt: When long frames (size > FIFO_SIZE) are transmitted, the number of remaining bytes after writing FIFO_SIZE bytes to the FIFO is stored in *TxCnt*.

The items described above are necessary to write/read the registers and to control the receiver and the transmitter of a channel. For the sake of security and clarity they are collected in a channel specific data link control structure.

4.3 ESCC2-Device Driver Module Entries

As mentioned before the ESCC2-device driver module is accessible by both, the device driver system and the ESCC2-device itself. The kind of access, however, is quite different.

The device driver system sends messages to the ESCC2-device driver module via the message entry point. On the other hand the ESCC2-device accesses to the driver module by entering an interrupt entry point. The two kinds of entry points are discussed below.

4.3.1 The Message Entry Point

Starting the device driver system all modules will be initialized. In particular, the ESCC2-device driver module is initialized by executing Escc2Init (CIM_MSG_DESCR_PTR cmd). To send messages to each other the modules of the device driver system use the module identification for defining the Source or Destination of a message in its corresponding header.

All messages, which are dedicated to the ESCC2-device driver module, reach the defined message entry point when Escc2MsgEntry (CI_MAILBOX far* message) is called by the device driver kernel.

To send a message to the device driver module is the same as setting the module a task. At the moment seven different types of tasks are supported by the ESCC2-message entry point. These tasks are distinguished by a task or message identification, named as **ID**, see **figure 11**.

The following message **IDs** are supported:

ID	Action/Function
MODULE_INIT (= 0)	Escc2Init()
INIT_DATA_LINK (= 1)	InitDataLink()
ASSIGN_ADDRESS (= 2)	AssignAddress()
CTRL_LOOP (= 3)	CtrlLoop()
CTRL_BAUD_RATE (= 4)	CtrlBaudRate()
CTRL_LINE_CODE (= 5)	CtrlLineCode()
SEND_FRAME (= 6)	SendFrame()

An important job of the message entry point is to decide quickly which action has to be executed referring to the **ID** of the current task. So, for the sake of speed, a switch statement was implemented.

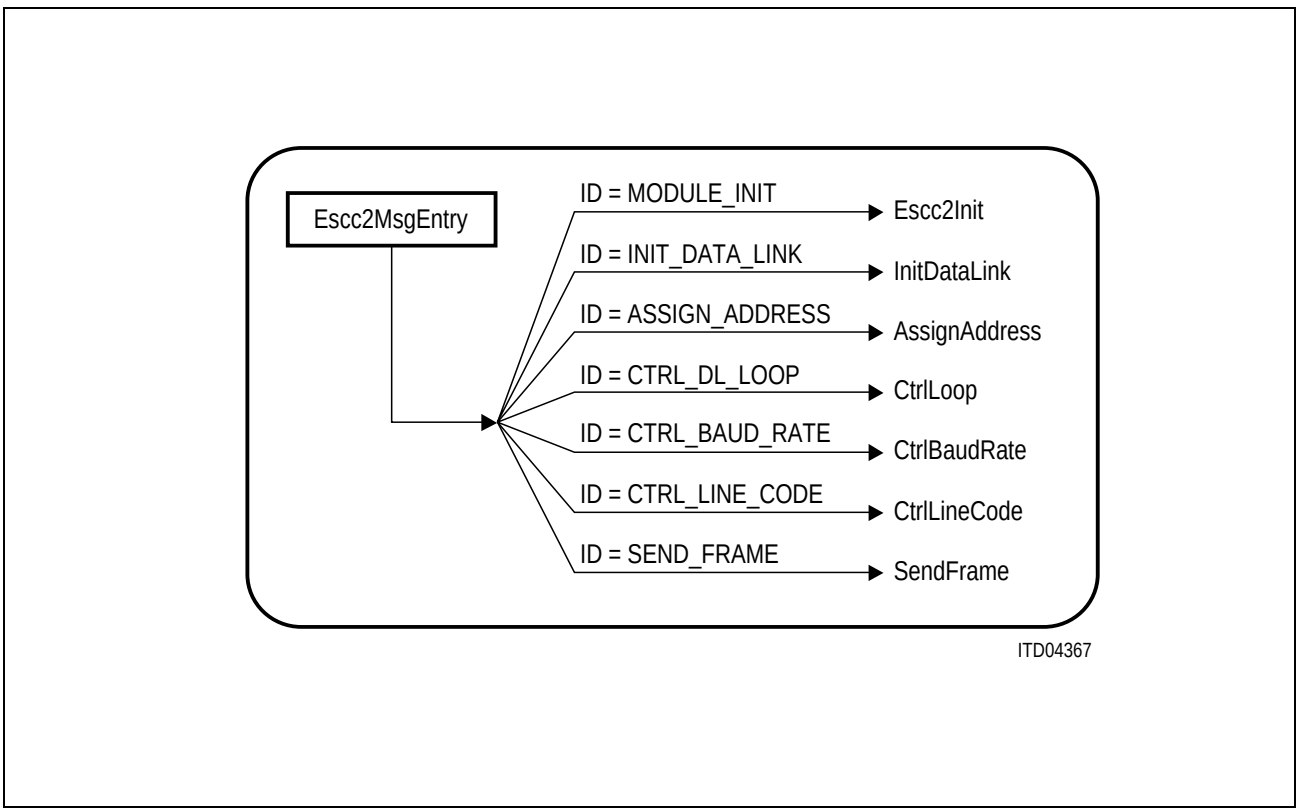


Figure 11
Message Entry Point

The functional description of the message entry point Escc2MsgEntry (CIM_MSG_DESCR_PTR cmd) is shown in **figure 11**.

The actions InitDataLink(), AssignAddress(), CtrlLoop(), CtrlBaudRate() and CtrlLineCode() are discussed in section 4.4.1. SendFrame() is explained in chapter 4.4.2.

4.3.2 The Interrupt Entry Point

Similar to the message entry point, the interrupt entry point is introduced during the initialization of the ESCC2-module.

As mentioned in the Technical Manual, special events in the ESCC2 are indicated by means of a single interrupt output with programmable characteristics (open drain, push-pull; IPC-register) which requests the CPU to read status information from the ESCC2, or, if interrupt mode is selected, to transfer data from/to ESCC2. Since only one INT-request output is provided, the cause of an interrupt must be determined by the CPU

– by evaluating the interrupt vector which is generated by the ESCC2 during an interrupt acknowledge cycle, and/or
– by reading the ESCC2's interrupt status registers (GIS, ISR0, ISR1, PIS).

In the device driver module the function of the interrupt output stage (pin INT) must be programmed as active high (push-pull output). Since the SAB 80C166 doesn't support the interrupt acknowledge cycle interrupt polling mode is used.

The channels Channel A and Channel B operate independently, so the base address and other control/status variables of a channel are collected in the DATA_LINK_CONTROL-structure, which is described in section 4.2.

As the basic address is known, the registers can be accessed. At first all interrupts should be masked out by writing FF$_H$ into the Interrupt Mask Register 0, 1 (IMR0, IMR1).

Further, as mentioned above, pin INT must be programmed as active high, so 03$_H$ has to be written to the Interrupt Port Configuration Register (IPC).

DdshSetHWIntVect (DEVICE_INT, Escc2Interrupt) installs the interrupt entry. DdshIntSetMode (DEVICE1_INT, MODE1) sets the interrupt mode: edge trigger mode. DdshIntCtrl (DEVICE1_INT, INT_ON) enables the interrupt (for more information about the DDSH-interface see chapter 3.5).

The interrupt function really can be seen as an entry to the core of the ESCC2-device driver module, which mainly consists of a transmitter and a receiver. A more detailed description of these two parts will follow in section 4.4.2. Obviously the interrupt entry point makes it possible, for the receiver or transmitter to be are ordered directly to read data from the receive FIFO or write data to the transmit FIFO.

After entering the interrupt function Escc2Interrupt(), the General Interrupt Status Register is analyzed and the channel and/or mode specific interrupt service routine is executed: ServeHdlcInterrupt() or ServeAsyncInterrupt(). During the execution of these functions the Interrupt Status Register is read. After that, the action corresponding to the specific kind of interrupt is executed (e.g. HdlcTxAction […] […] ()).

The arrays of function pointers … Action […] […] () are described in section 4.4.2. INTRET releases the interrupt logic of the SAB 80C166 to be able to accept further interrupts.

To summarize, up to now the entry points of the ESCC2-module have been described. There are two different kinds of entry points:

– the message entry point Escc2MsgEntry (…), which branches into functions corresponding to message **IDs**,
– the interrupt entry point, that calls functions according to the interrupt indication.

4.4 General Architecture of the ESCC2-Device Driver Module

Now, that the entry points have been described, the structure of the ESCC2-module can be explained. Besides the entry points the ESCC2-device driver module can be divided into two parts:

- Initialization and Control Routines
- Receiver and Transmitter

4.4.1 Initialization and Control Routines

By manipulating the parameters of the actual command the user or an application module (HACVT in this application) is able to control the initialization routines via the message entry point. Since there are two channels, one can select the corresponding channel by modifying the structure element *Entity*.

Six initialization and control routines are supported:

Escc2Init (CIM_MSG_DESCR_PTR *cmd*) initializes the ESCC2-device driver module. The base device address is set by executing DdshGetDeviceAddr (DEVICE1). For the sake of speed during the access to the general interrupt status register after an interrupt has occured the address of this registers is stored in *GisPtr*. Furthermore all arrays and some DATA_LINK_CTRL-structure elements are initialized in Escc2Init(). An important task finally is the initialization of the interrupt entry (see section 4.3.2).

InitDataLink (CIM_MSG_DESCR_PTR *cmd*) contains the general initialization of all ESCC2-mode specific functions and data. Per default the access to the register which are **common** to all modes (HDLC, ASYNC, …), is done via the Hdlc_Wr or Hdlc_Rd union element of the corresponding DATA_LINK_CTRL-structure (Interrupt Mask Register, Interrupt Status Register, Command Register, …).

Parameter 1 of the CIM_MSG_DESCR-structure accessible by the pointer *Command*, selects the transfer mode (HDLC, ASYNC, …). Parameter 2 has a different meaning in HDLC-mode (p2 = transparent, auto-mode …) and ASYNC-mode (p2 = termination character). The clock mode is represented by parameter 3. By writing 60_H into the Mode Register (MODE) for example the HDLC non-auto-mode and a 16-bit address field are selected. Furthermore the channel configurations are fixed by programming the channel configuration registers. During the execution of InitDataLink() the interrupt masks are set to enable the RME-, RFO-, RPF-, XDU- and XPR-interrupts in HDLC-mode or the TCD-, RFO-, RPF- and XPR-interrupts in ASYNC-mode. Finally the InitDataLink-function resets the receiver and transmitter.

AssignAddress (CIM_MSG_DESCR_PTR *cmd*) writes the low address value into the Receive Address Byte Low Register 1 (RAL1) and the high address value into the Receive Address Byte High Register 1 (RAH1). These values can be used for LAPD 2-byte address field recognition corresponding to the ESCC2 non-auto-mode. Parameter 1 of the CIM_MSG_DESCR-structure contains the high address value, parameter 2 contains the low address value. This function is used only in HDLC-mode.

CtrlBaudRate (CIM_MSG_DESCR_PTR cmd) controls the baud rate of the channel in clock mode set during InitDataLink() by programming the Baud Rate Generator Register (BGR) and the Channel Configuration Register 2 (CCR2). Parameter 1 of the CIM_MSG_DESCR-structure contains the baud rate division factor.

CtrlLineCode (CIM_MSG_DESCR_PTR cmd) determines the line code used for data transmission by writing corresponding values into the Channel Configuration Register 0 (CCR0). Four line codes are defined: NRZ, FM0, FM1 and Manchester. Parameter 1 of the CIM_MSG_DESCR-structure determines which line code is selected.

CtrlLoop (CIM_MSG_DESCR_PTR cmd) switches test loops in transmission path on and off by manipulating the least significant bit in the Mode Register (MODE). To switch the loop on or off depends on the value of parameter 1 of the CIM_MSG_DESCR-structure.

4.4.2 Receiver and Transmitter

As mentioned in section 4.2.2 the receiver and the transmitter are the core of the ESCC2-driver module. Indeed they are the most important parts of the device driver module, because they are responsible for a high speed data communication. Therefore, they have to guarantee a quick service of the interrupts generated by the ESCC2.

During the development of the receiver and the transmitter a further point has to be considered: the FIFO-size of 32 bytes for both the receiver and the transmitter. Because of the FIFO-size, for example, two cases have to be distinguished:

1) to receive/transmit a frame with a length less than or equal FIFO-size,

2) to receive/transmit a frame with a length greater than FIFO-size.

In the first case, a RME-interrupt (Receive Message End) is received in HDLC-mode; the transmitter on the other hand has to write XTF (transmit pool full) and XME (transmit message end) to the command register after writing data to the transmit FIFO. In ASYNC-mode a TCD-interrupt (Termination Character Detected) is received; the transmitter has to write XF (transmit Frame) to the command register.

In the second case, however, a RPF-interrupt (Receive Pool Full) is received. The receiver is informed through this interrupt that the FIFO accessible to the processor does not contain the end of the frame, the receiver is in a state which can be described as *receiving*. The transmitter, in the second case, has to write XTF (transmit pool full) to the command register, after writing the first part of the frame to the transmit FIFO. The transmitter has to keep in mind, that the transmission of the current frame is not finished yet; the transmitter is in a state which can be described as *sending*.

This short introduction shows that various events may occur on the one hand in the ESCC2-device, as indicated by interrupts (RPF, RME, TCD, XPR, …), and on the other hand on application side, requesting to send a short or long frame. Further it should be noticed that the receiver and the transmitter stay in certain states, waiting for the next event. Finally, if the event occurs, the receiver or transmitter executes some action corresponding to the actual state and actual event.

Event State Table – General Aspect

The method of working described above can be represented by an event state table, shown in **figure 12**.

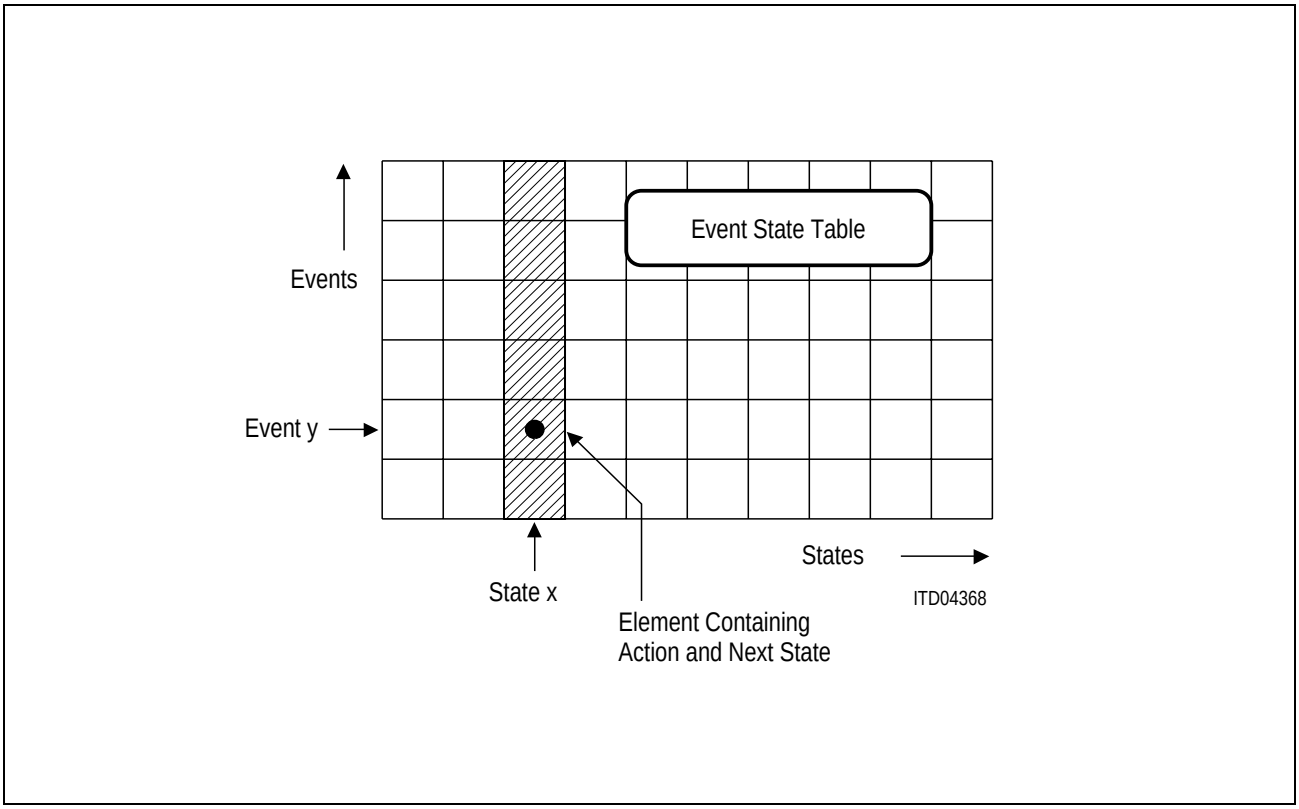


Figure 12
State Event Table

Each element of this table represents a certain action to be executed and a next state to be taken on – corresponding to the actual state and actual event. The execution of these actions and state transitions is controlled by the finite state machine. This technique allows immediate execution of actions by avoiding the use of switch statements. The actions become very short and easier to read. The following chapter gives an idea of the concept of a finite state machine.

As mentioned before the two channels of the ESCC2 operate independently. To minimize line of code the channels share the same action code. This is a further advantage of the finite state machine, since the channel only has to retain its current state. At the occurence of an event related to this channel, the channel's state and the event cause the corresponding action. The code of this action is independent from the actual channel. The description of the channel's state is an important part of the control structure *DataLinkCtrl [channel]*, introduced above.

A more detailed description of the actions themselves can be given by SDL (specification and description language) diagrams. They are useful for describing source code, since they allow a quick overview of the algorithm.

Finite State Machines and SDL-Diagrams

A key concept used in many driver and protocol models is the finite state machine, illustrated in **figure 13**.

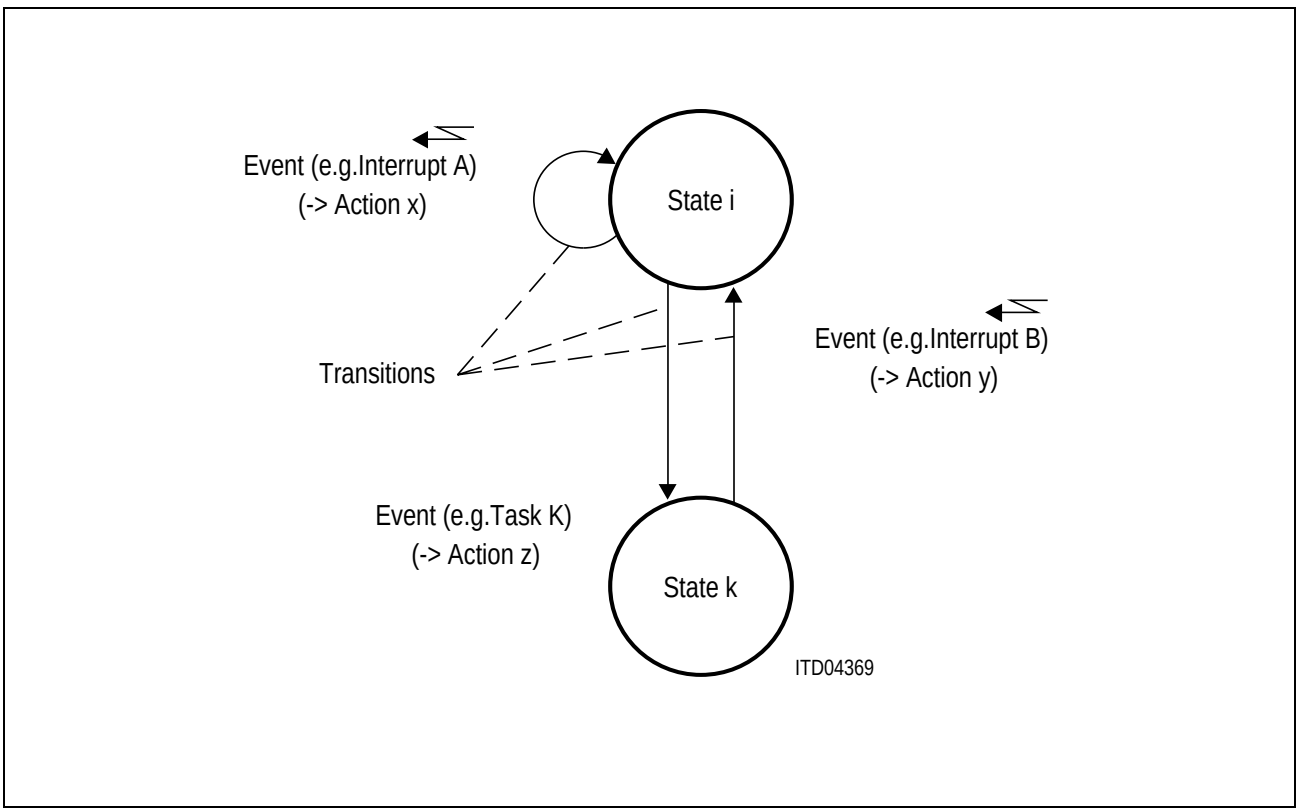


Figure 13
Finite State Machine

With this technique, each driver machine (i.e. transmitter or receiver) is always in a specific state at every instant of time. From each state, there are zero or more possible transitions to other states. Transitions occur when some event takes place. Given a complete description of the driver machines, it is possible to draw a directed graph showing all the states as nodes and all the transitions as directed lines or arcs. One particular state, for example, is designated as the initial state (e.g., called IDLE). This state corresponds to the description of the system when it starts running. From the initial state some, perhaps all, of the other states can be reached by a sequence of transitions.

Receiver – Event State Table

The receiver is suitable to make oneself familiar with the concept of event driven programming. Assuming HDLC-mode there are three interrupts controlling the receiver: RPF-interrupt, RME-interrupt and RFO-interrupt. The RPF-interrupt means that 32 (= FIFO-size) bytes were received, but the message end has not yet been received.

The RME-interrupt, however, means that 32 bytes or less than 32 bytes were received and the message end has been received, too. The RFO-interrupt says that a Receive Frame Overflow has occured. Depending on the current state these events cause different actions. In the following table

the event and state combination for the receiver are shown (different states correspond to different columns, different events to different lines).

Event	**State**	
	IDLE	RECEIVING
RFO	ServeRfo	RpfServeRfo
INTERRUPT	IDLE	IDLE
RME	ServeRme	RpfServeRme
INTERRUPT	IDLE	IDLE
RPF	ServeRpf	RpfServeRpf
INTERRUPT	RECEIVING	RECEIVING

The actions of the receiver are described in a locally defined two-dimensional array, called *HdlcRcAction […] […] (…)*.

In ASYNC-mode instead of an RME-interrupt a TCD-interrupt occurs; so the ServeRme () function can be replaced by ServeTCD (). This function is the only difference to the treatment of HDLC-mode.

During the initialization at first *HdlcRcAction/AsyncRcAction […] […]* is set to *NoAction* for all elements of the arrays. Then, corresponding to the event state table the action is entered only for the event-state-combination, which may occur.

…

HdlcRcAction [IDLE] [RME_INTERRUPT] = ServeRme;

…

AsyncRcAction [IDLE] [TCD_INTERRUPT] = Serve Tcd;

…

The assigned control structure contains elements like the actual state of the receiver *RcState* and the pointer to the current position *RcCurr* to which the next bytes (of a frame longer than 32 bytes) have to be written.

Receiver – Finite State Machine

Interpreting the event state table of the receiver, the finite state machine can be constructed without problems. **Figure 14** shows the graph, which describes the receiver in the ESCC2-module as a finite state machine.

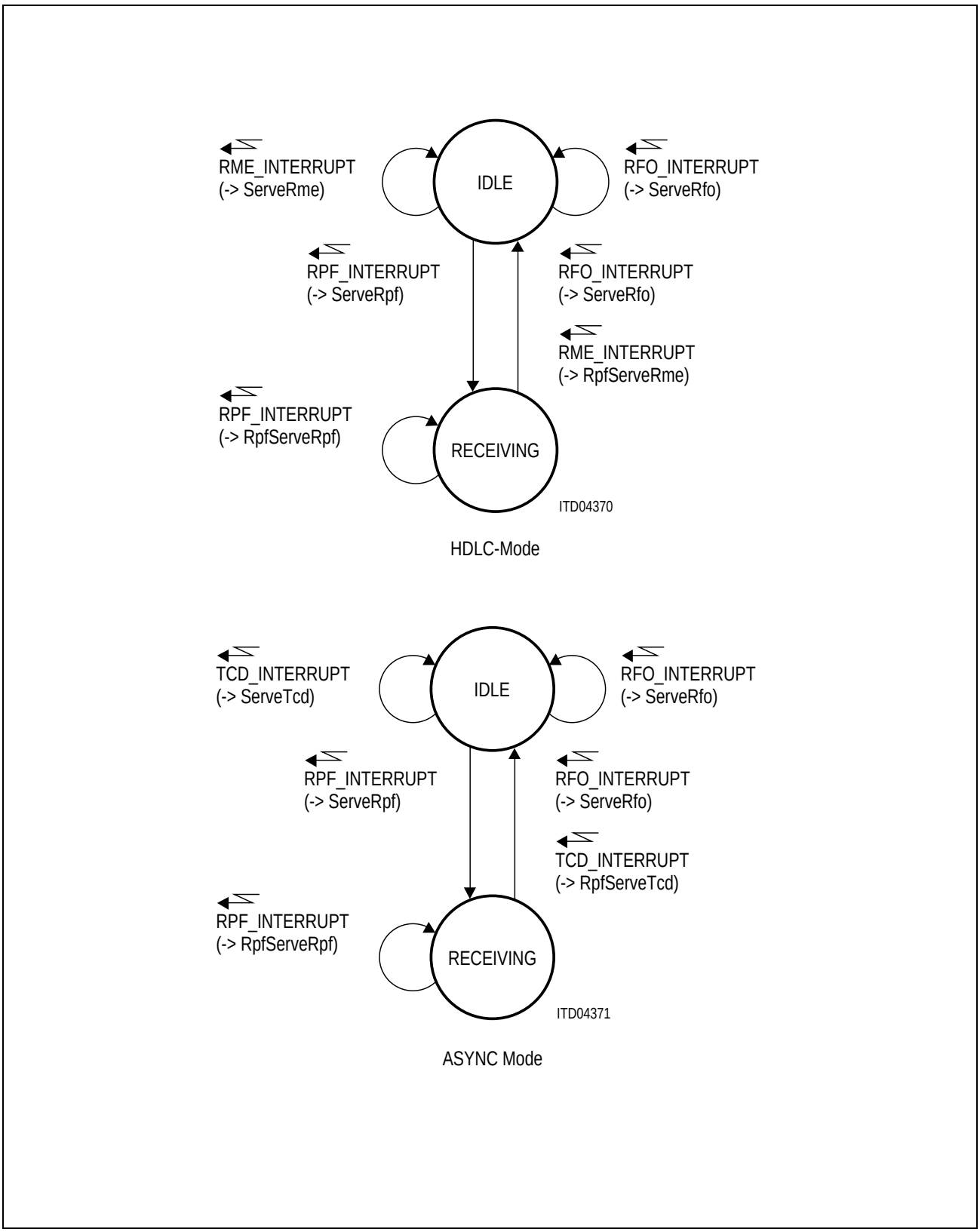


Figure 14
Receiver Finite State Machine

Receiver – SDL-Diagrams

All SDL-diagrams start with the current state, followed by the actual event.

Getting an RPF-interrupt during the state IDLE means that the receiver has to allocate a data buffer greater than 32 (= FIFO-size) bytes. Then the first 32 bytes of the frame, read from the RFIFO can be written to the buffer. After that the RFIFO is reset by writing RMC to the command register. The pointer to data buffer *RcCurr* has to be increased by FIFO-size. Finally the state is changed.

If the actual state is RECEIVING when a RPF-interrupt occurs, the same algorithm (as for state: IDLE, event: RPF_INTERRUPT) is executed except for allocating the data buffer.

If an RME-(TCD-) interrupt occurs, at first the length of the received frame is read from the RBC-register. The action ServeRme (ServeTcd) has to allocate a data buffer with less or equal then 32 bytes. From now the action RpfServeRme (RpfServeTcd) and ServeRme (ServeTcd) are similar.

The receive status byte is read from the RFIFO as the first byte after the end of the received frame. Then analyzing the receive status byte it has to be decided if the received frame is ok. If the frame is ok, a message is sent to the user, otherwise the function ReportRcError() is executed, to give more detailed error information.

In ASYNC-mode there is no receive status byte available if the RFDF-bit in the RFC-register is set to 0.

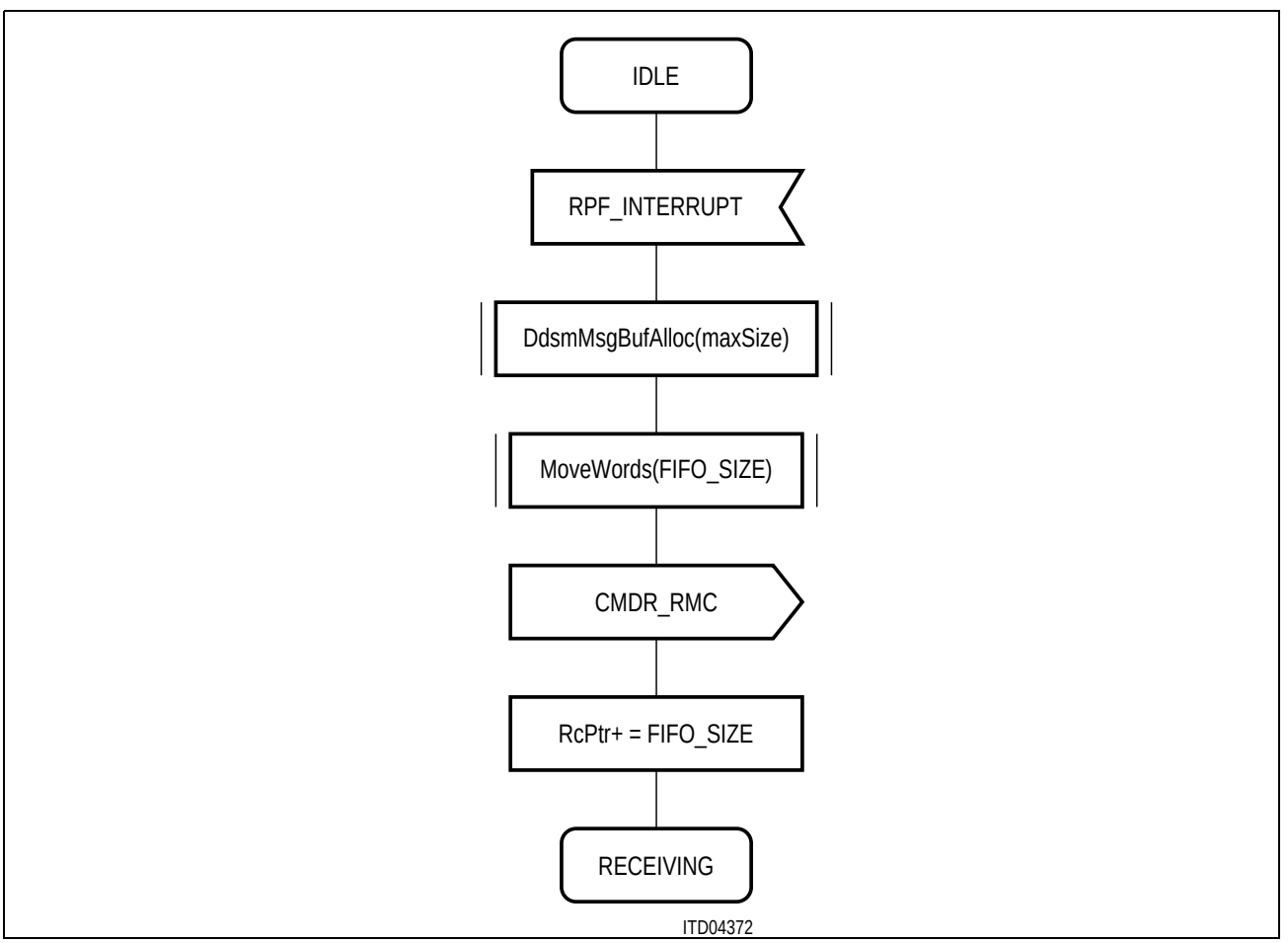


Figure 15
Action: *ServeRpf*

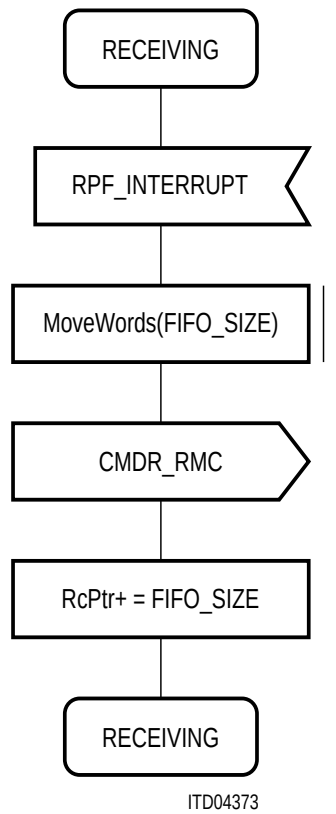


Figure 16
Action: *RpfServeRpf*

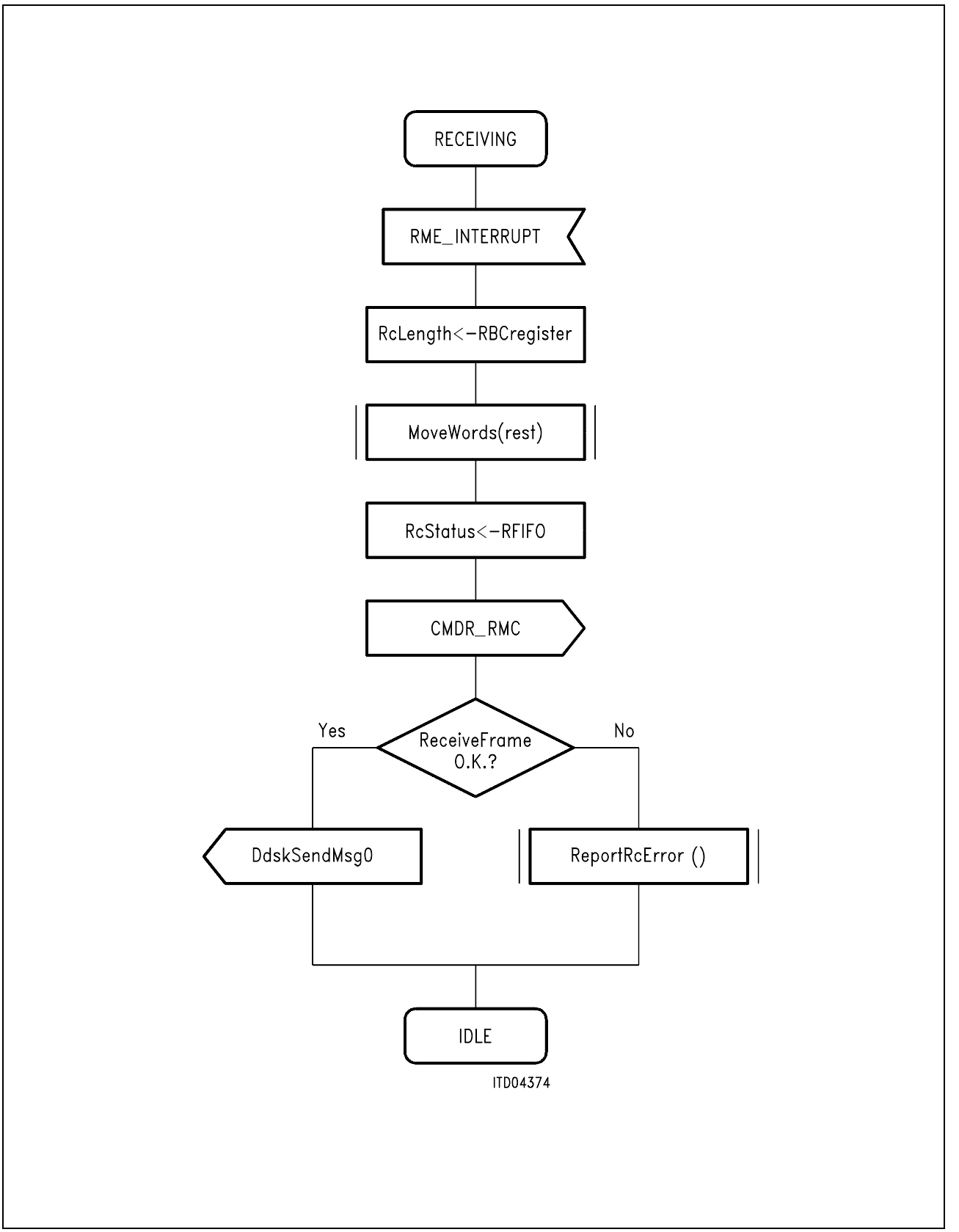


Figure 17
Action: *RpfServeRme*

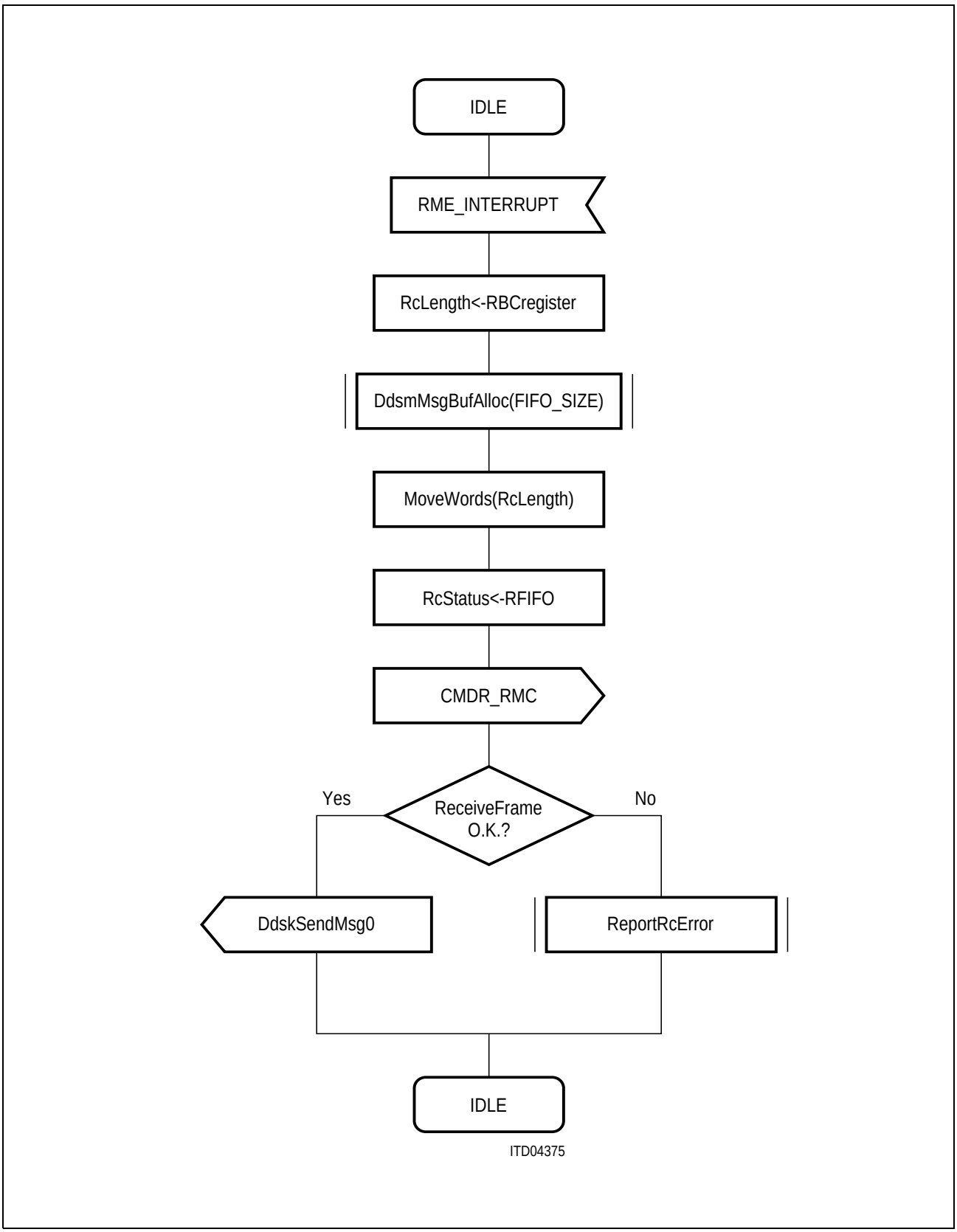


Figure 18
Action: *ServeRme*

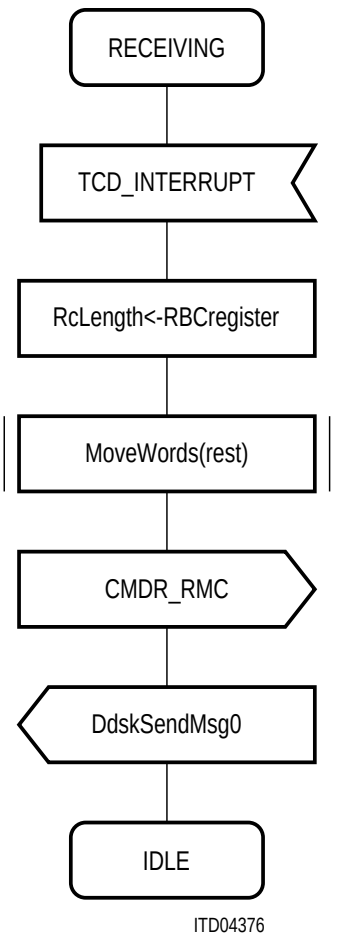


Figure 19
Action: *RpfServeTcd*

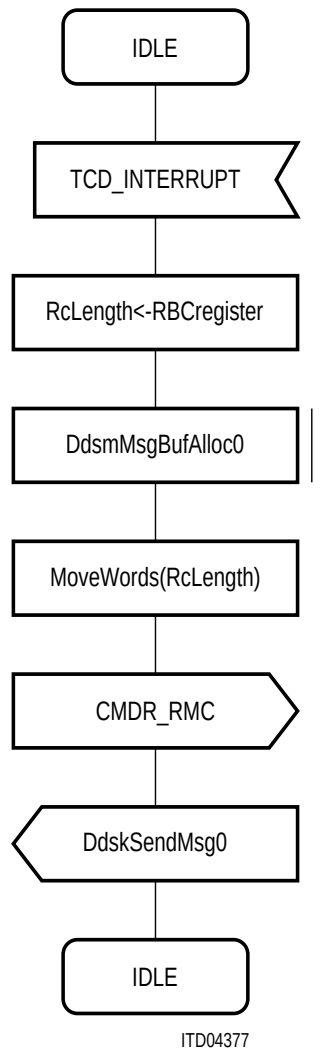


Figure 20
Action: *ServeTcd*

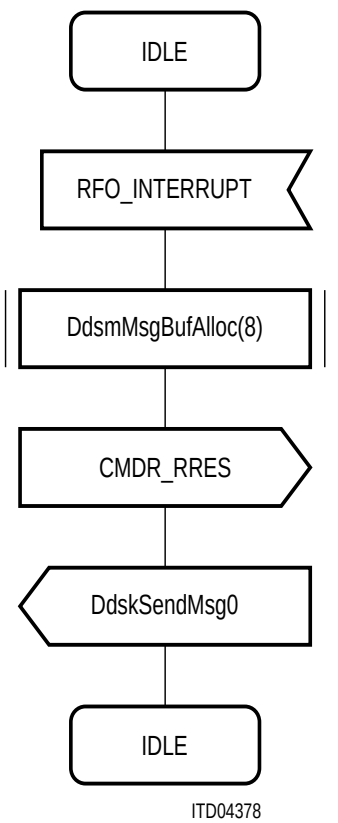


Figure 21
Action: *ServeRfo*

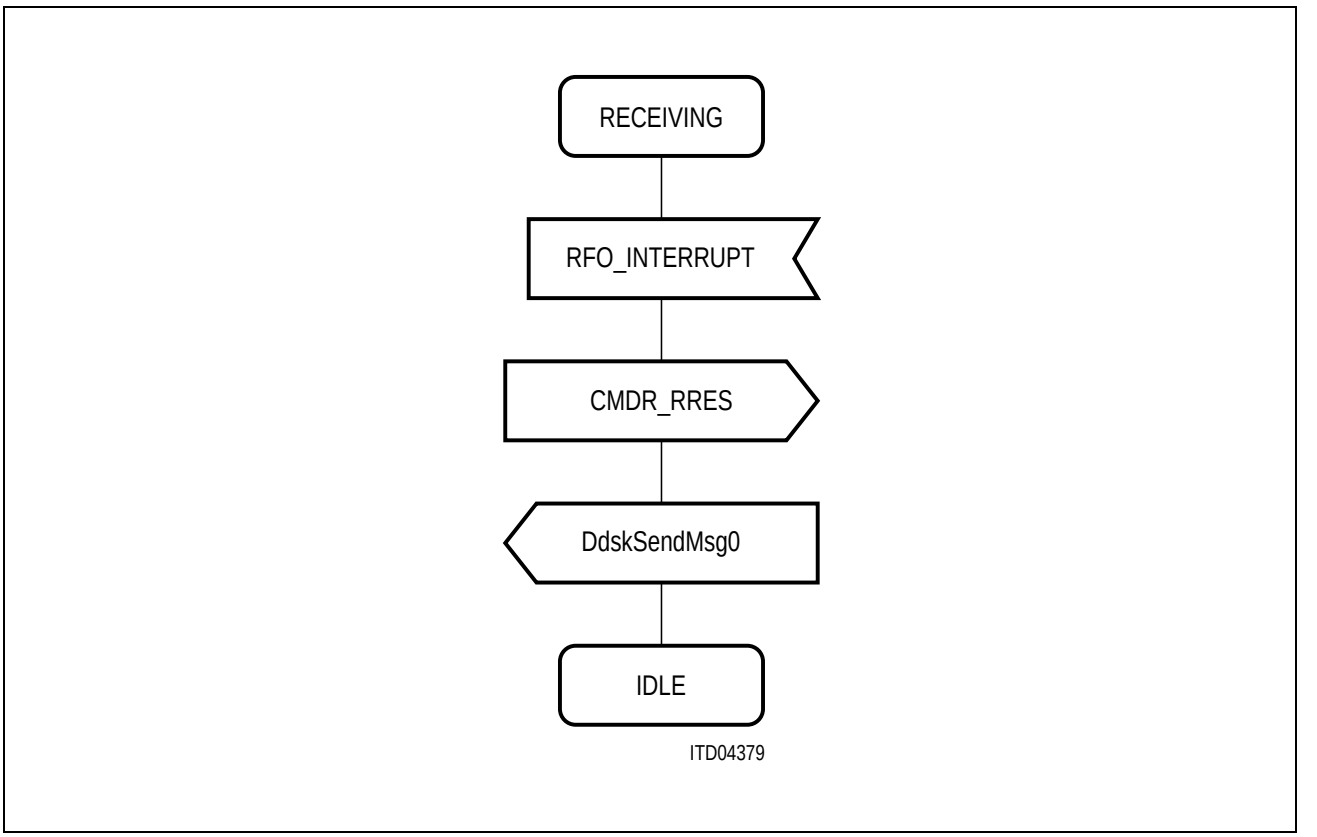


Figure 22
Action: *RpfServeRfo*

Transmitter – Event State Table

While the events for the receiver are generated by interrupts only, the events for the transmitter are caused by both the application and the ESCC2-device. A key position on application side has the function SendFrame (…). Corresponding to the length of the frame (shorter or longer than FIFO-size), two different events are generated: SEND_SHORT_FRAME, SEND_LONG_FRAME.

After transmit FIFO reset the ESCC2-channel gets an XPR-interrupt. This event brings the transmitter from the state IDLE into the state READY. The transmitter is able to start transmission.

It may happen that the transmitter gets a task while it is still busy with a previous one. Consecutive tasks are queued in a linked list.

Because of that, the control structure of the channel contains the following two elements:

.
.

CIM_MSG_DESCR_PTR head;
CIM_MSG_DESCR_PTR tail;

.

For clarity's sake the event table is divided into two parts. The first part shows the events caused by the user. Different states corresponds to different lines, the events are represented by the columns.

State	Event	
	SEND_SHORT_FRAME	**SEND_LONG_FRAME**
SENDING	NoAction	NoAction
	SENDING	SENDING
SENDING_END	NoAction	NoAction
	SENDING_END	SENDING_END
READY	TxShortFrame	StartTxLongFrame
	SENDING_END	SENDING
IDLE	NoAction	NoAction
	IDLE	IDLE

The device generates two interrupts dedicated to the transmitter: XDU-, XPR-interrupt.

State	Event	
	XPR-INTERRUPT	**XPR-INTERRUPT**
SENDING	ContTxLongFrame	ServeXdu
	SENDING_END [1)]	IDLE
SENDING_END	ServeNextFrame	ServeXdu
	READY [2)]	IDLE
READY	NoAction	NoAction
	READY	IDLE
IDLE	SetReady	NoAction
	READY	IDLE

1) SENDING_END is only be taken on, if no bytes remain. Otherwise the state will be unchanged.

2) READY is only be taken on, if the task queue is empty. Otherwise the next task (long or short frame) will be executed and the state will be SENDING or SENDING_END.

In ASYNC-mode no XDU-interrupt occurs. This is the only difference to the treatment of HDLC-mode.

During the initialization at first *HdlcTxAction/AsyncTxAction […] […]()* is set to *NoAction* for all elements of the arrays. Then, corresponding to the event state table the Action is entered only for the event-state-combination which can occur.

…
HdlcTxAction[IDLE][XPR_INTERRUPT]=SetReady;
…
AsyncTxAction[IDLE][XPR_INTERRUPT]=SetReady;
…

Transmitter – Finite State Machine

The following graphs show the transmitter of the ESCC2-driver module as a finite state machine, for both HDLC-mode and ASYNC-mode.

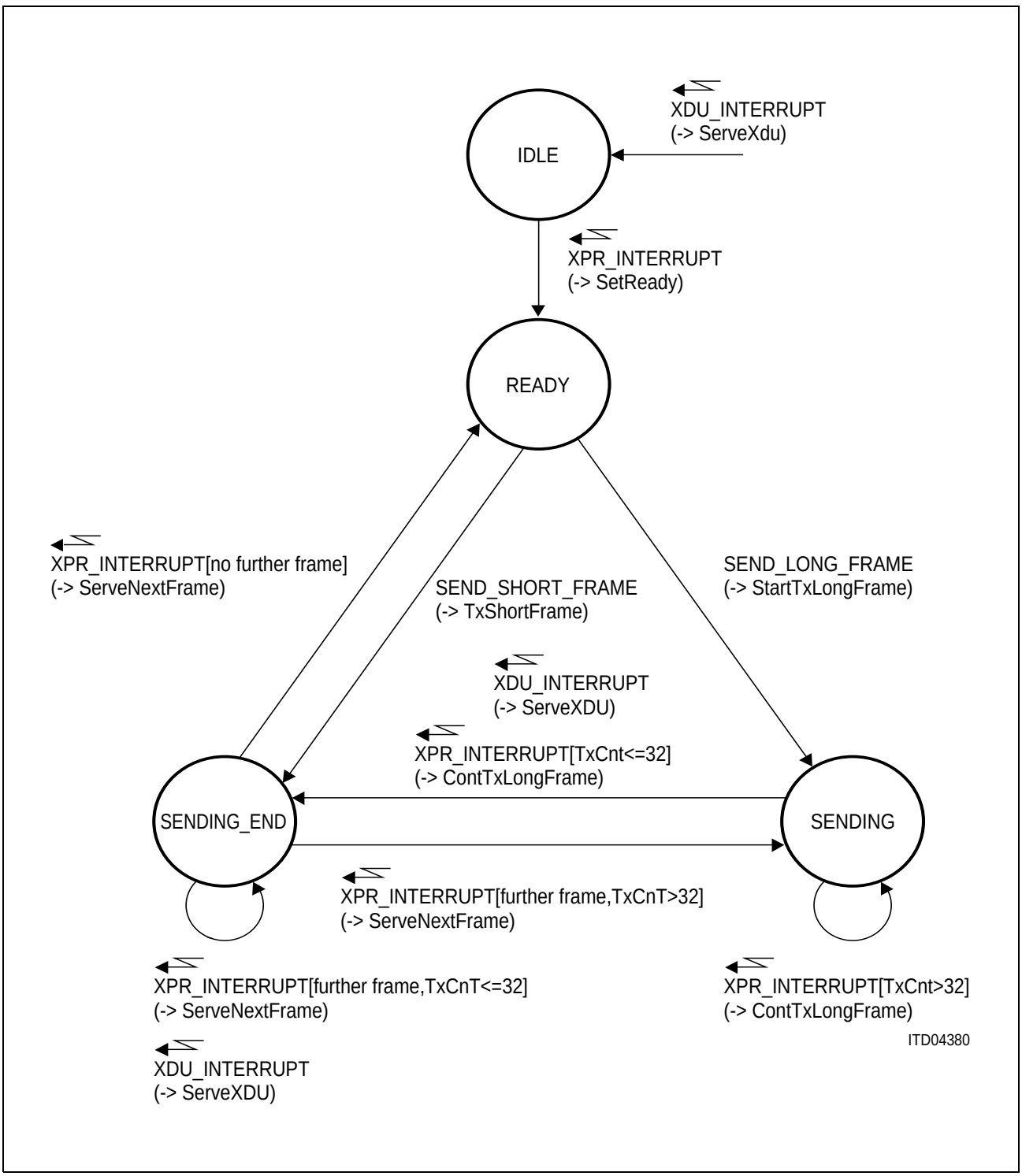


Figure 23
Transmitter Finite State Machine HDLC-Mode

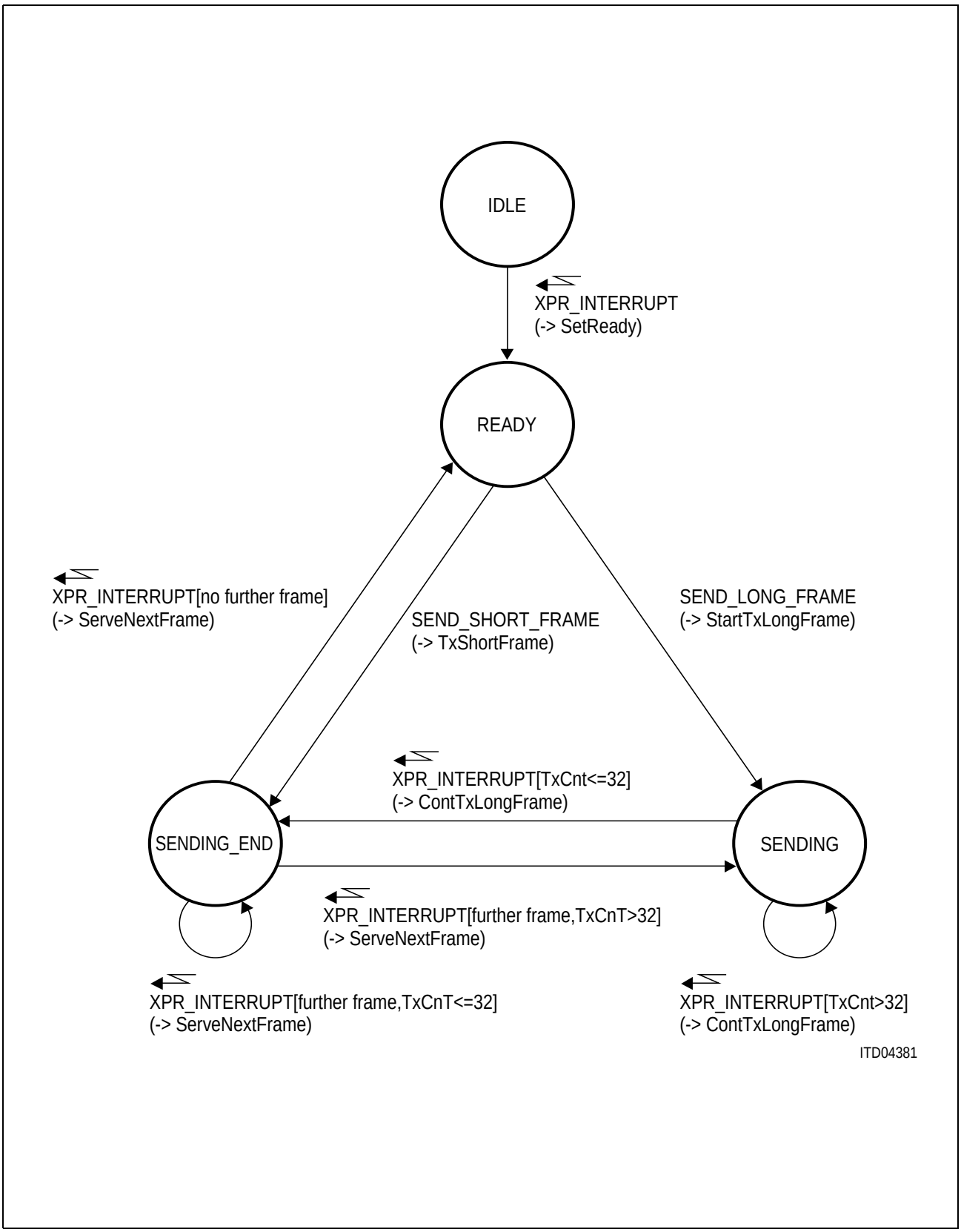


Figure 24
Transmitter Finite State Machine ASYNC-Mode

Transmitter – SDL-Diagrams

At the end of this chapter the SDL-diagrams for the transmitter are presented. They provide a brief overview of the transmitter's actions.

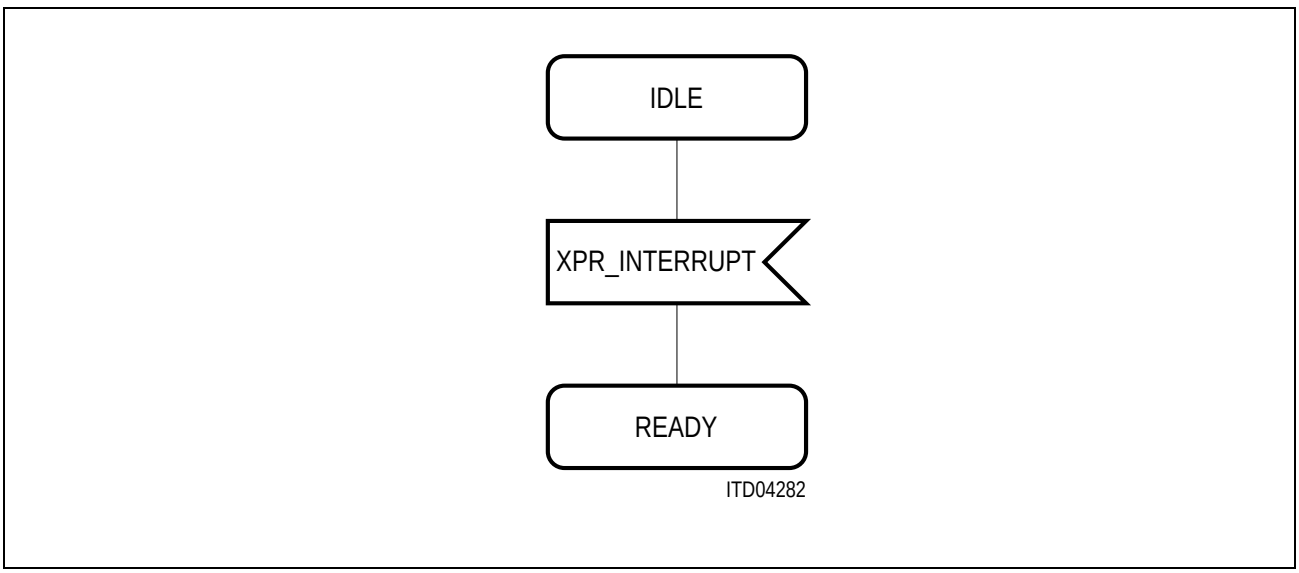


Figure 25
Action: *SetReady*

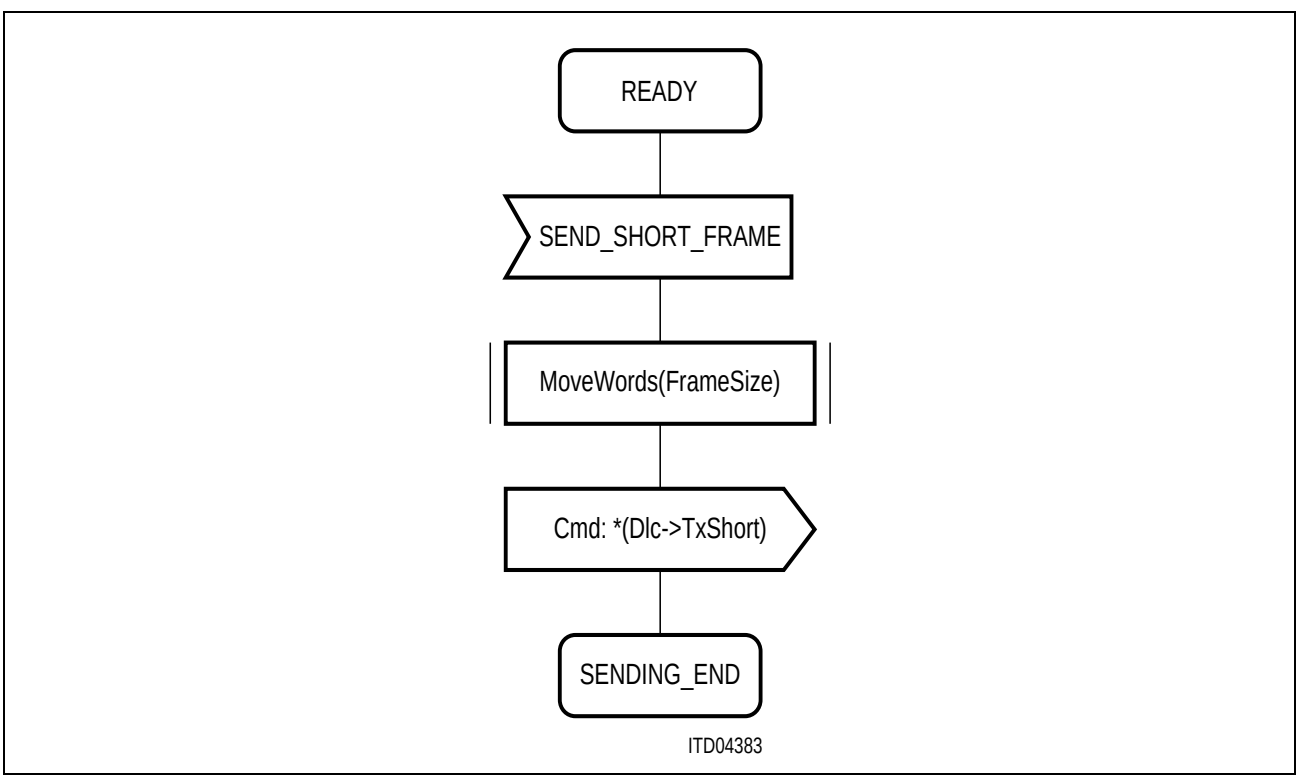


Figure 26
Action: *TxShortFrame*

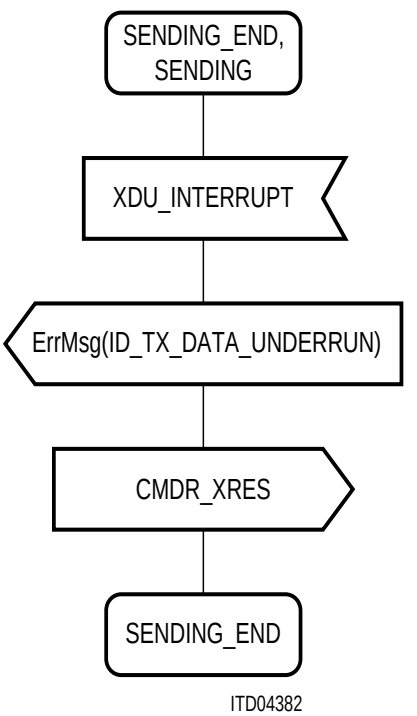


Figure 27
Action: *ServeXdu*

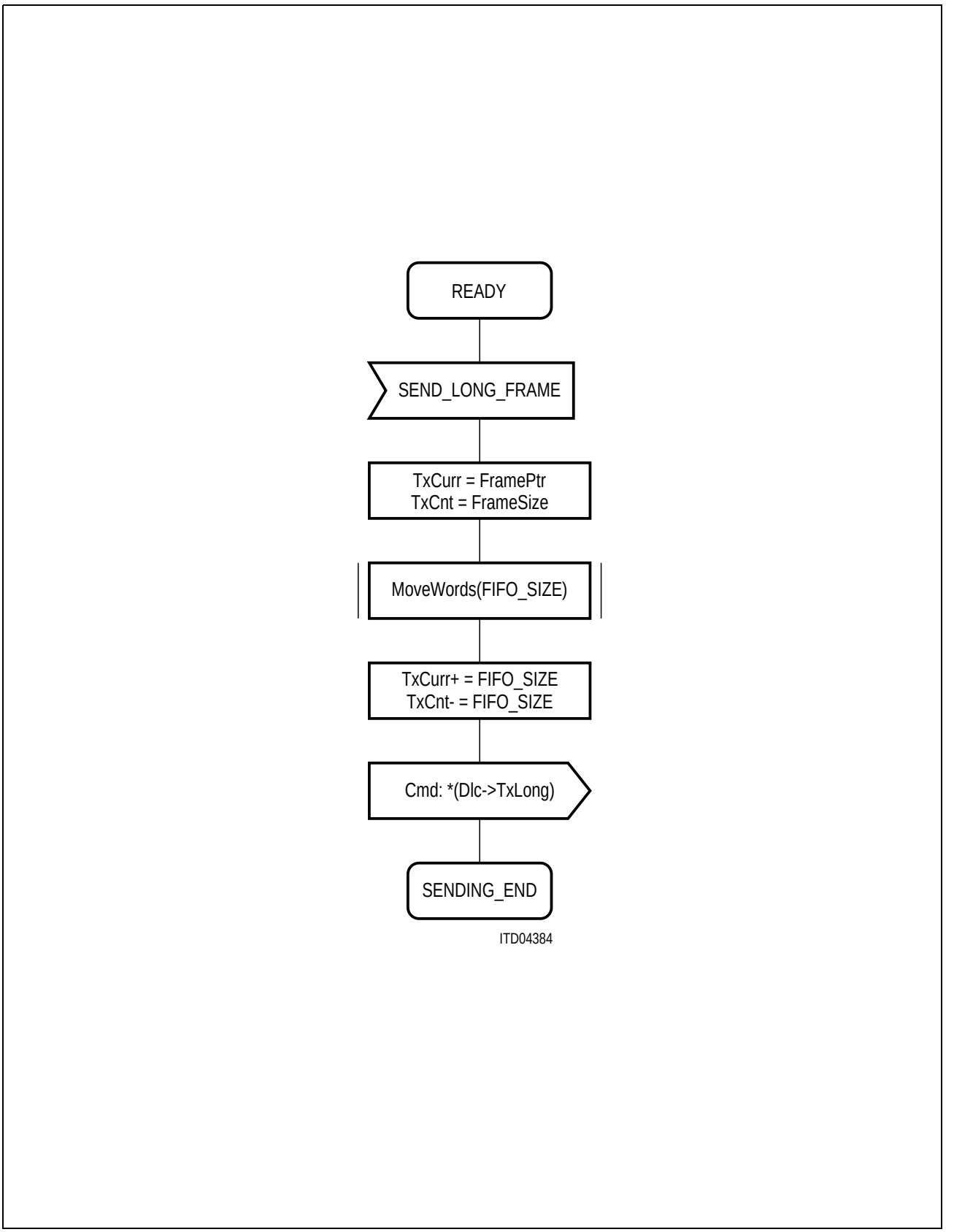


Figure 28
StartTxLongFrame

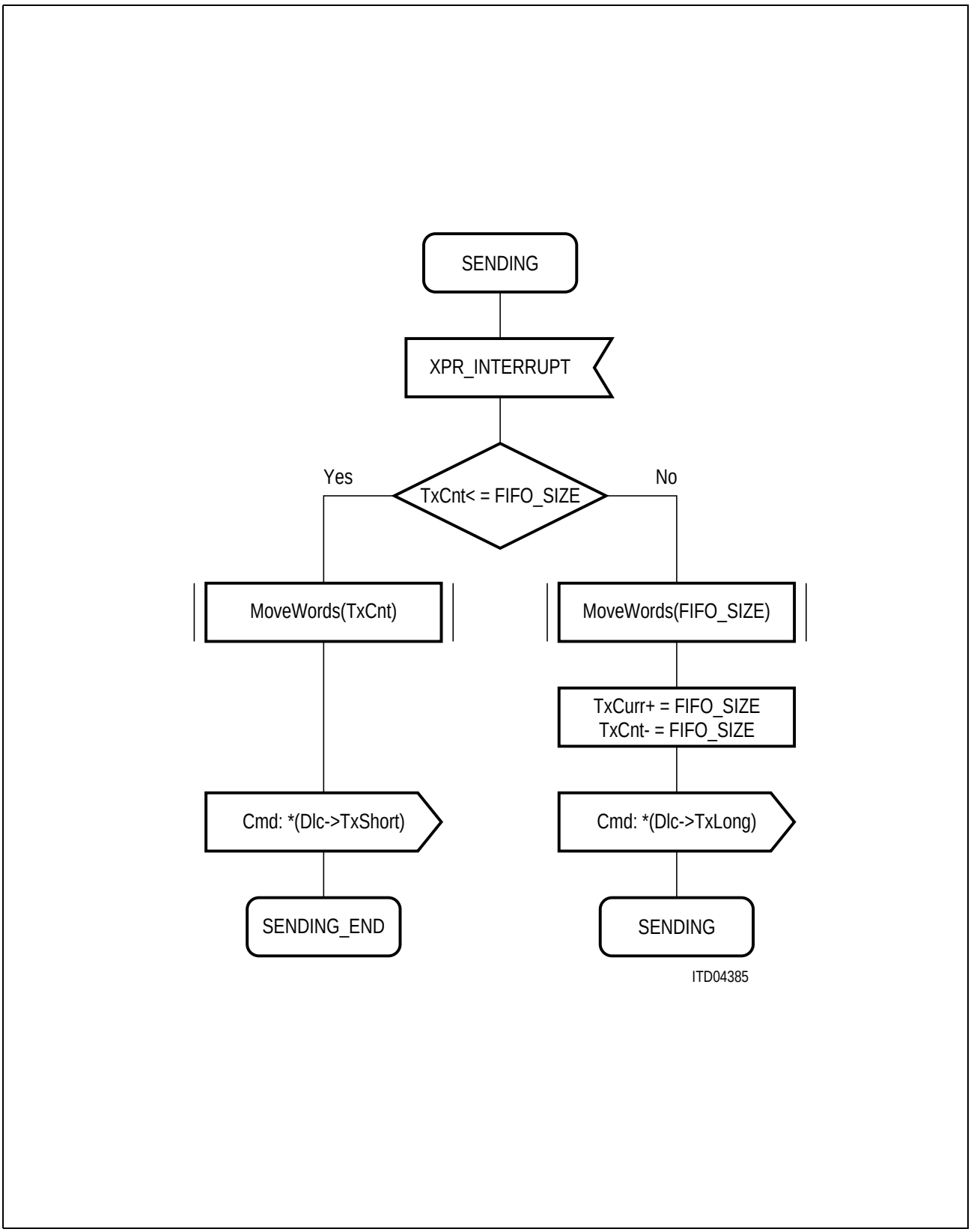


Figure 29
Action: *ContTxLongFrame*

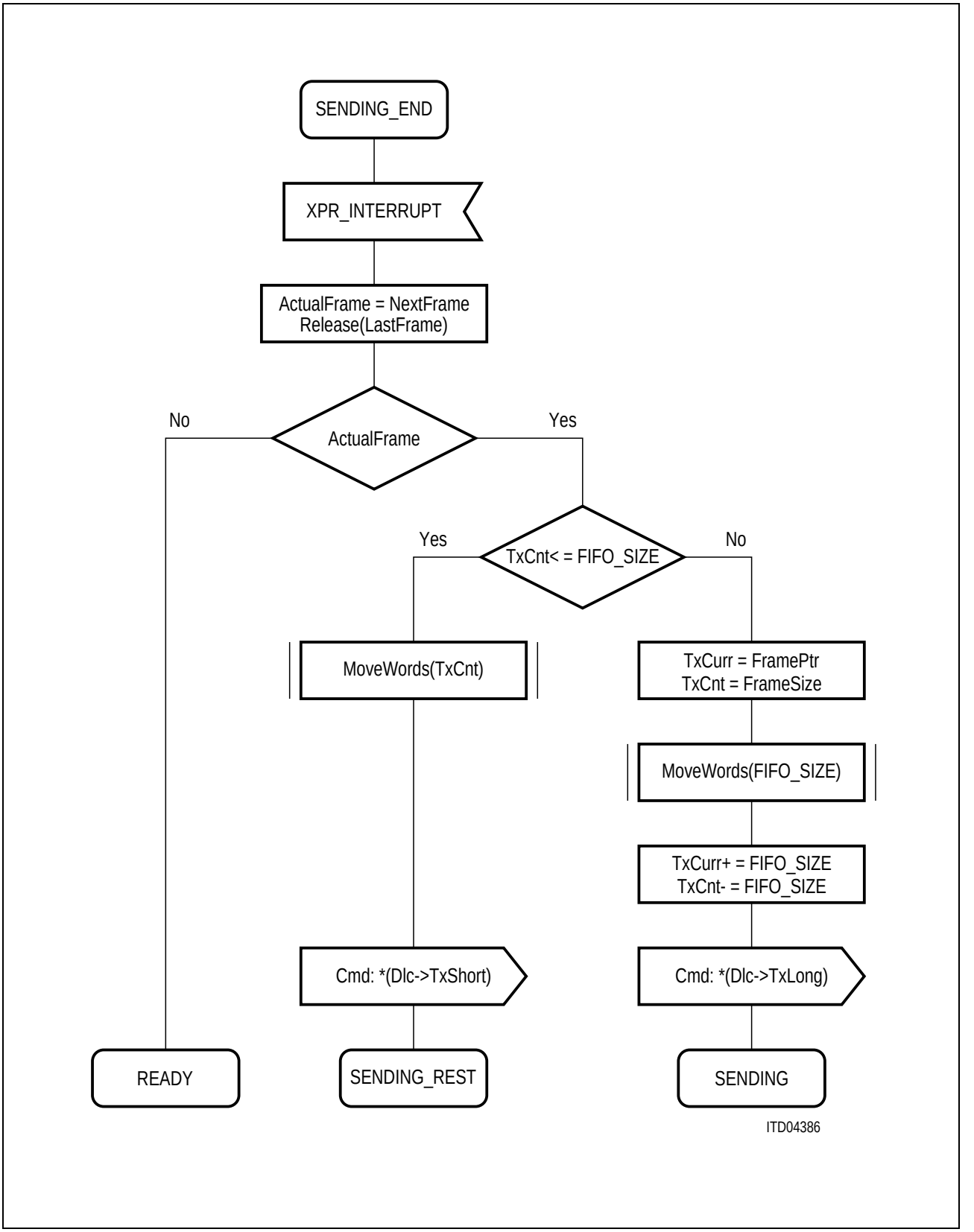


Figure 30
Action: *ServeNextFrame*

5 The Application Module HACVT

HACVT (HDLC- to ASYNC-ConVerTer) converts HDLC-frames received to ASYNC-frames and vice versa. Since the ESCC2 is able to transfer frames in HDLC-mode as well as in ASYNC-mode and the ESCC2-device driver supports these transfer modes the application module HACVT can be designed very easily.

5.1 Structure of the Application Module

Depending on the specific application, all modules used are integrated by executing DdsIntegrate() (Refer to chapter 3.2). DdsIntegrate() sets the message entry points of the modules which are necessary to run the application. The message entry points are set by executing DdskSetMsgEntryPoint (INT16 module, MSG_FCT_PTR_msg_entry).

The application on hand sets the message entry points to the ESCC2-device driver module and to the application module HACVT itself. After the Device Driver System (DDS) has executed DdsIntegrate() all modules are initialized by the DDS (Escc2Init(), HaCvtInit()).

5.2 Detailed Description of the Application Module

In addition to the function DdsIntegrate() and the message entry point HaCvtMsgEntry (CIM_MSG_DESCR_PTR command) the application module contains three routines, which initialize the module or perform the conversion of the transfer mode: TransCvtInit (…), HdlcToAsync (…) and AsyncToHdlc (…).

HaCvtInit (CIM_MSG_DESCR_PTR command) sends message to the ESCC2-device driver module to initialize the data links (ID = INIT_DATA_LINK). The channel identification (entity), transfer mode (P1), receive mode or termination character (P2) and clock mode (P3) are passed as parameters of the message. After the initialization the application module receives messages from the ESCC2-device driver whenever a frame has been received.

There are two kinds of message IDs which are supported: RC_HDLC_FRAME_ESCC2_OK and RC_ASYNC_FRAME_ESCC2_OK.

If a HDLC-frame has been received the function HdlcToAsync (CIM_MSG_DESCR_PTR cmd) is executed; if an ASYNC-frame has been received AsyncToHdlc (CIM_MSG_DESCR_PTR cmd) is executed.

HdlcToAsync (…): All HDLC-frames received on one channel (e.g. ESCC2-channel A) are sent transparently in ASYNC-mode on the other channel (e.g. ESCC2-channel B). The HDLC-frames are completed with the termination character, to guarantee correct transfer in ASYNC-mode. The address byte are not stripped from the HDLC-frame.

AsyncToHdlc (…): All ASYNC-frames received on one channel (e.g. ESCC2-channel B) are sent transparently in HDLC-mode on the other channel (e.g. ESCC2-channel A). From the ASYNC-frame received the termination character (e.g. AA_H) is stripped.

Appendix A

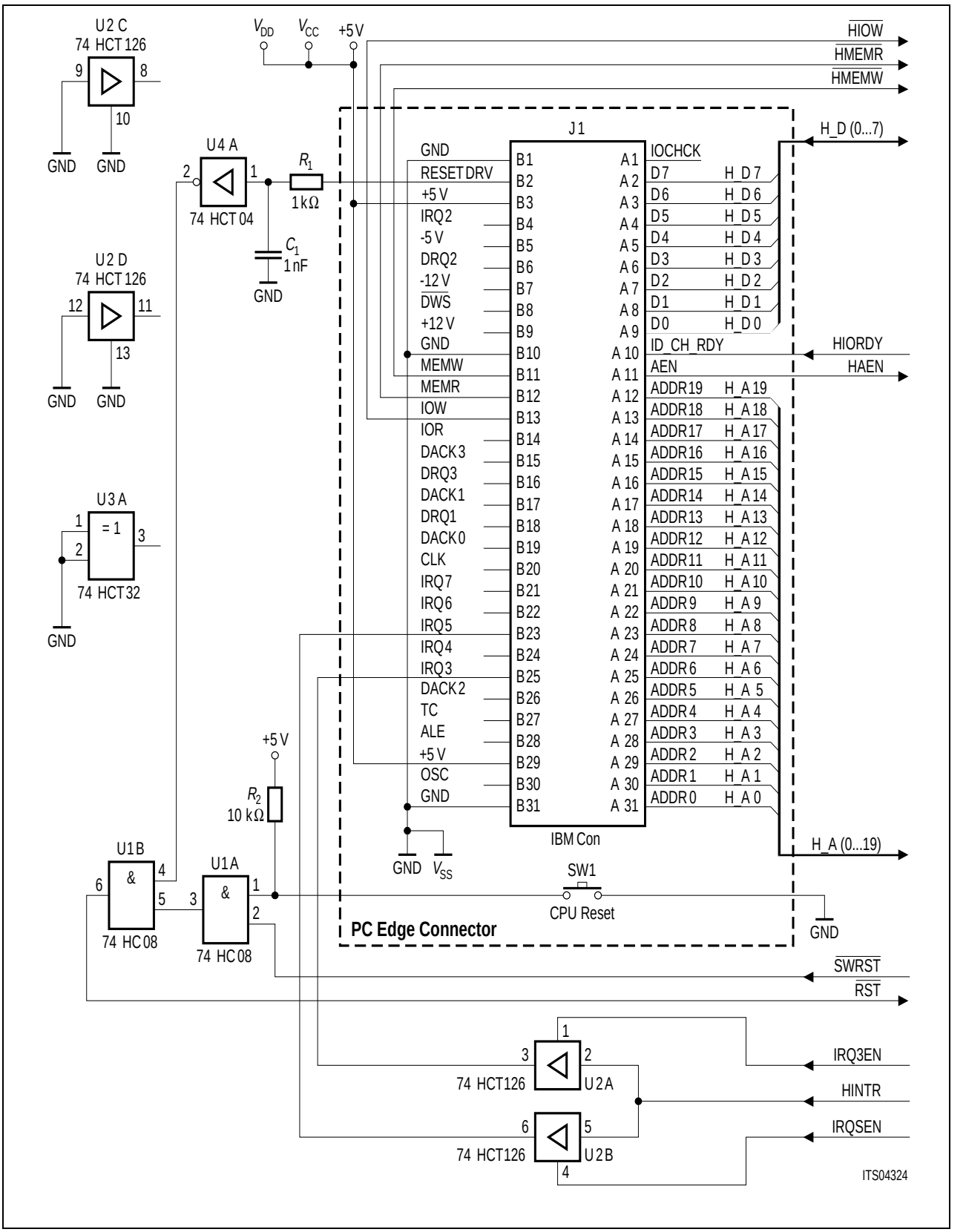


Figure 31
PC-Host Bus

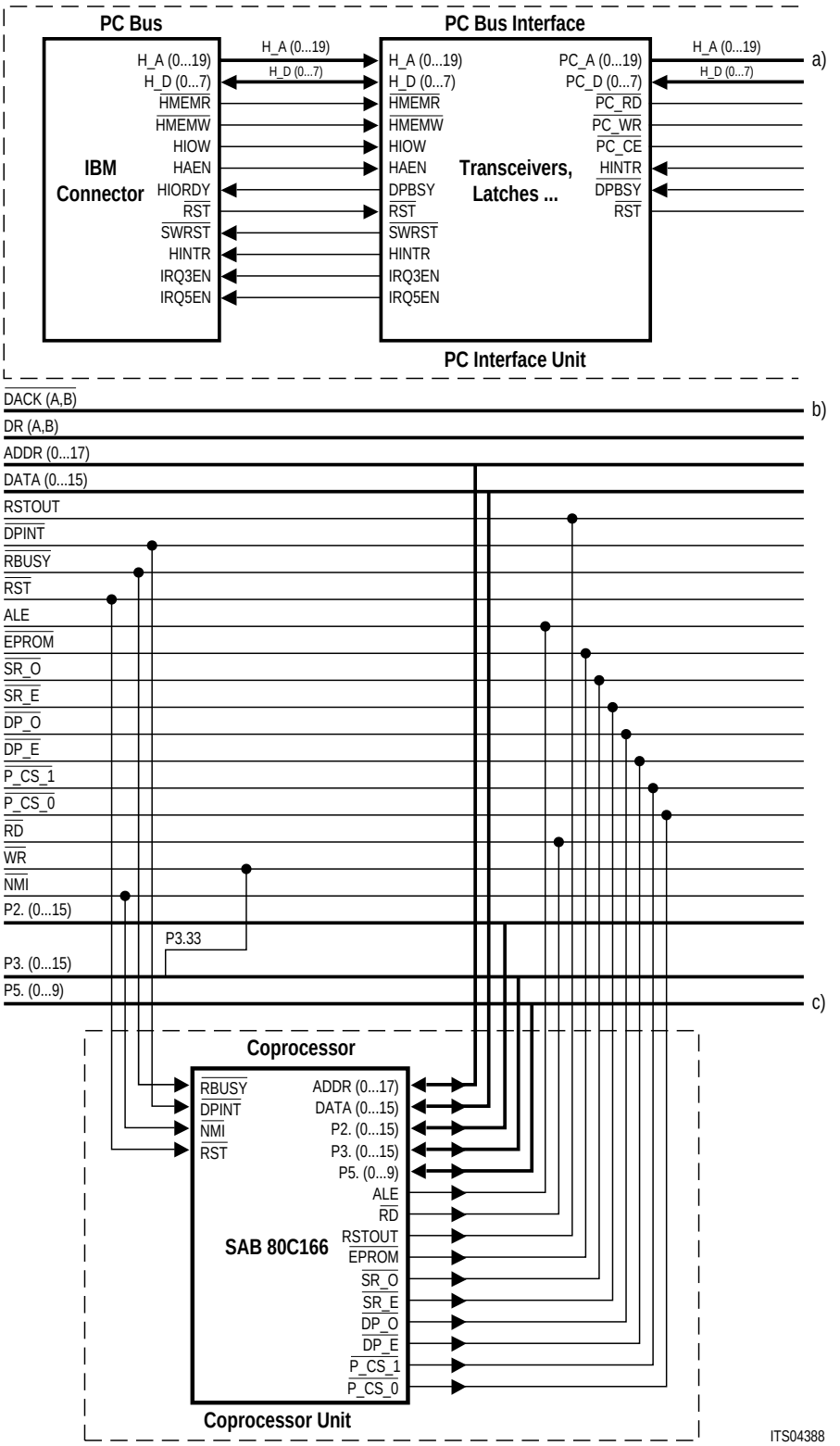
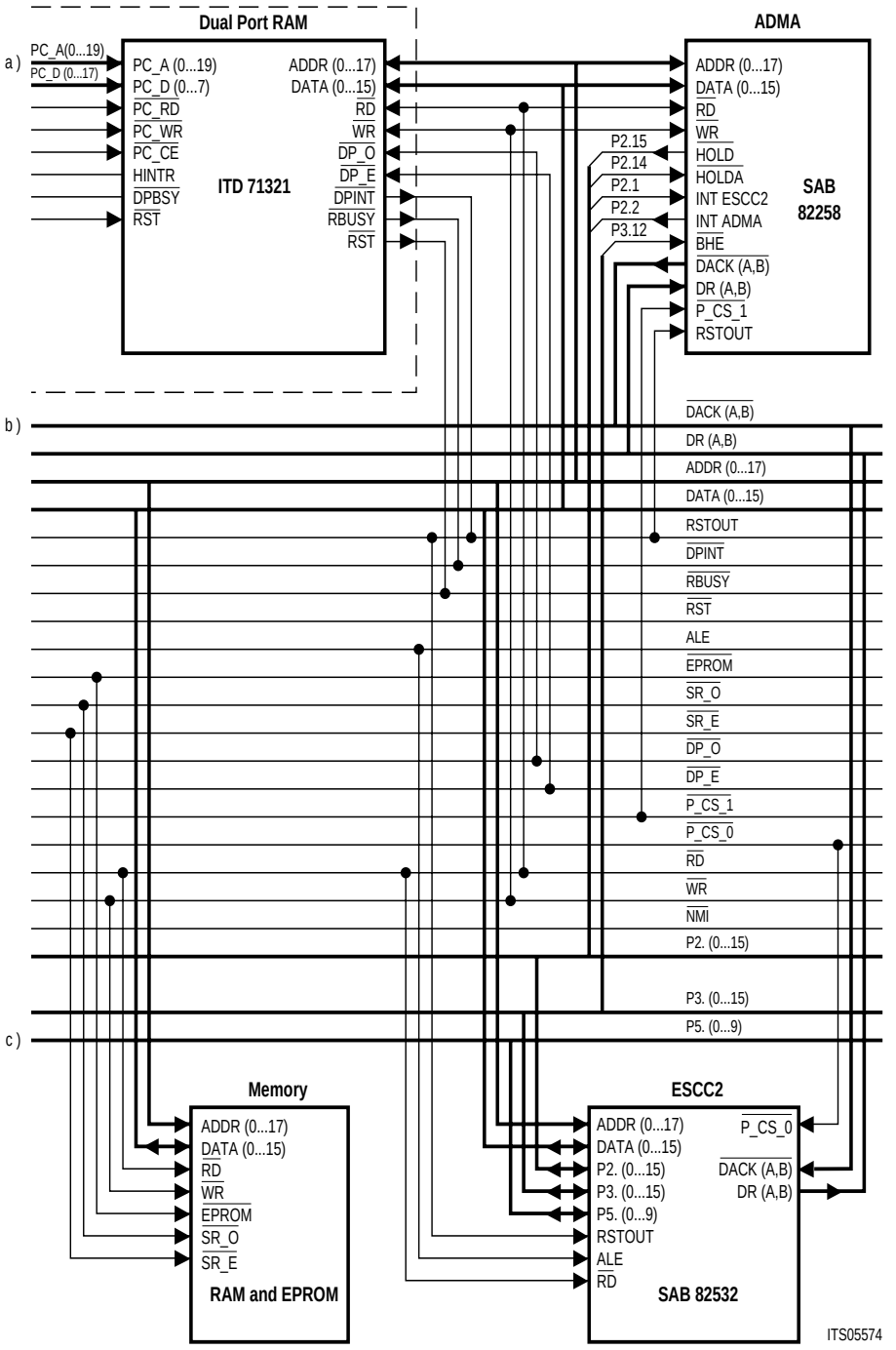


Figure 32
Main Block Diagram


Dual Port RAM
ADMA
a)
PC_A(0...19)
PC_D (0...17)
PC_A (0...19)
PC_D (0...7)
PC_RD
PC_WR
PC_CE
HINTR
DPBSY
RST
ADDR (0...17)
DATA (0...15)
RD
WR
DP_O
DP_E
DPINT
RBUSY
RST
ITD 71321
ADDR (0...17)
DATA (0...15)
RD
WR
HOLD
HOLDA
INT ESCC2
INT ADMA
BHE
DACK (A,B)
DR (A,B)
P_CS_1
RSTOUT
P2.15
P2.14
P2.1
P2.2
P3.12
SAB 82258
b)
DACK (A,B)
DR (A,B)
ADDR (0...17)
DATA (0...15)
RSTOUT
DPINT
RBUSY
RST
ALE
EPROM
SR_O
SR_E
DP_O
DP_E
P_CS_1
P_CS_0
RD
WR
NMI
P2. (0...15)
P3. (0...15)
P5. (0...9)
c)
Memory
ADDR (0...17)
DATA (0...15)
RD
WR
EPROM
SR_O
SR_E
RAM and EPROM
ESCC2
ADDR (0...17)
DATA (0...15)
P2. (0...15)
P3. (0...15)
P5. (0...9)
RSTOUT
ALE
RD
P_CS_0
DACK (A,B)
DR (A,B)
SAB 82532
ITS05574

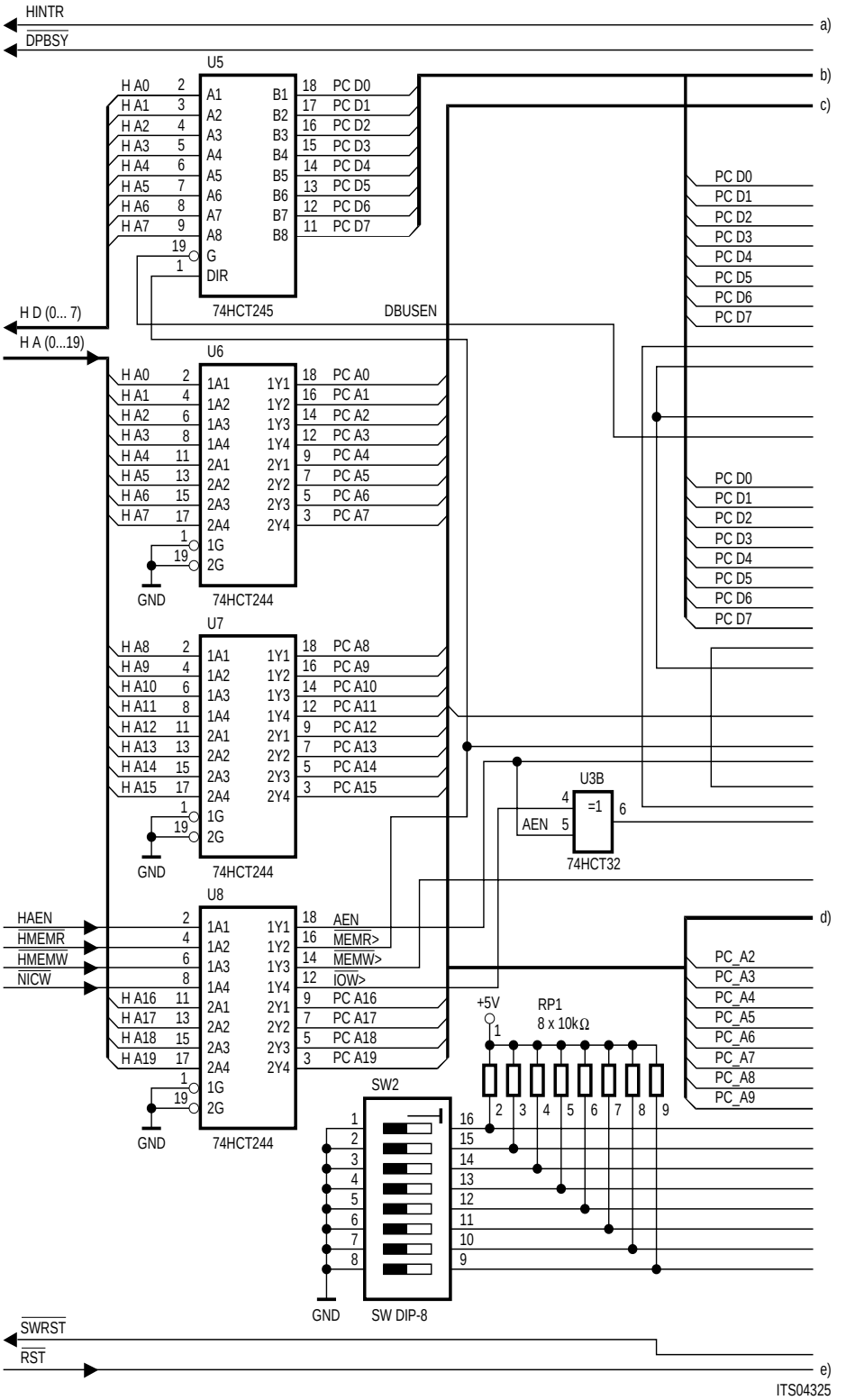
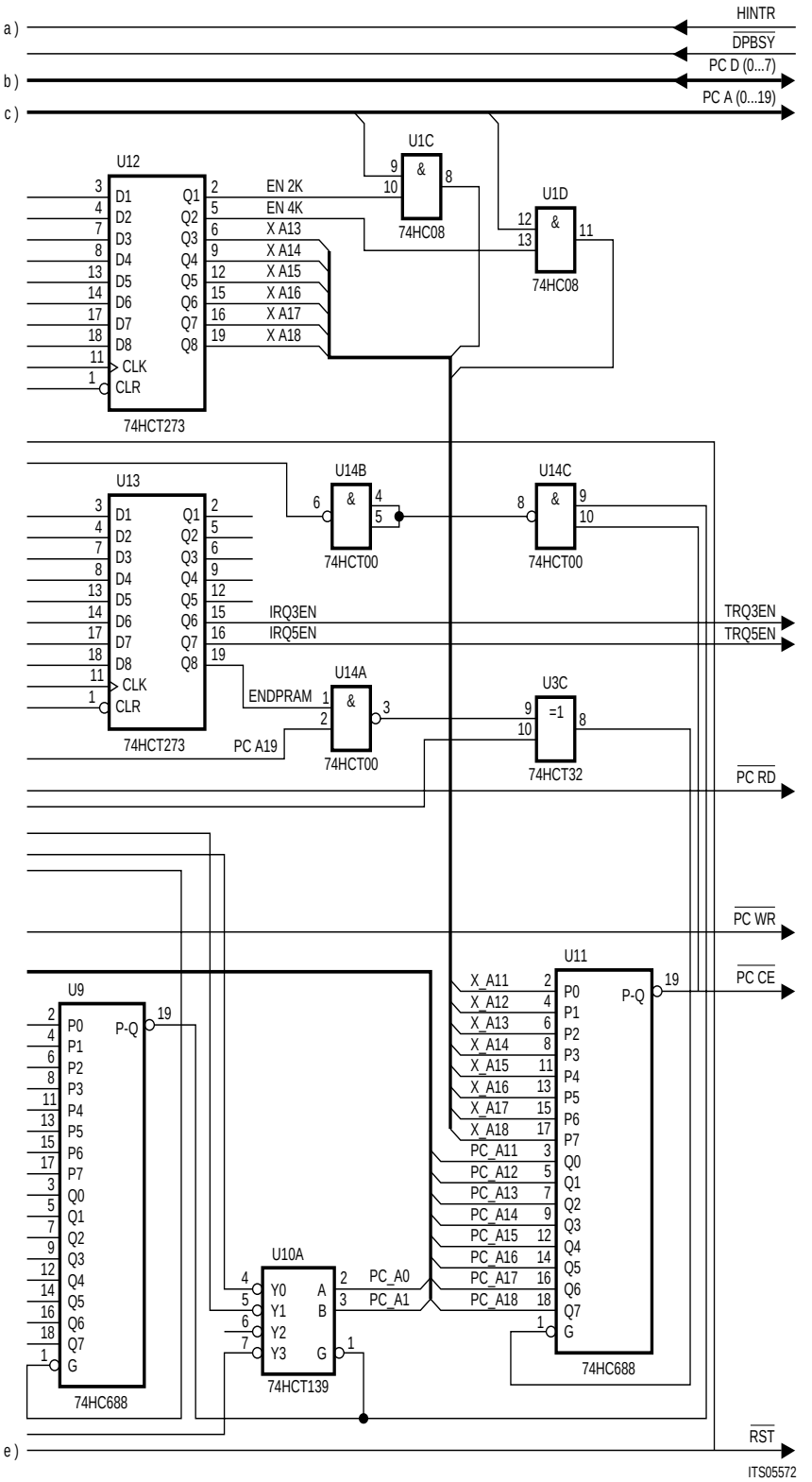


Figure 33
PC-Bus Interface-Logic


a)
b)
c)
e)
HINTR
DPBSY
PC D (0...7)
PC A (0...19)
U12
74HCT273
EN 2K
EN 4K
X A13
X A14
X A15
X A16
X A17
X A18
U1C
74HC08
U1D
74HC08
U13
74HCT273
U14B
74HCT00
U14C
74HCT00
IRQ3EN
IRQ5EN
TRQ3EN
TRQ5EN
U14A
74HCT00
ENDPRAM
PC A19
U3C
74HCT32
PC RD
PC WR
PC CE
U9
74HC688
U10A
74HCT139
PC_A0
PC_A1
U11
74HC688
X_A11
X_A12
X_A13
X_A14
X_A15
X_A16
X_A17
X_A18
PC_A11
PC_A12
PC_A13
PC_A14
PC_A15
PC_A16
PC_A17
PC_A18
RST
ITS05572

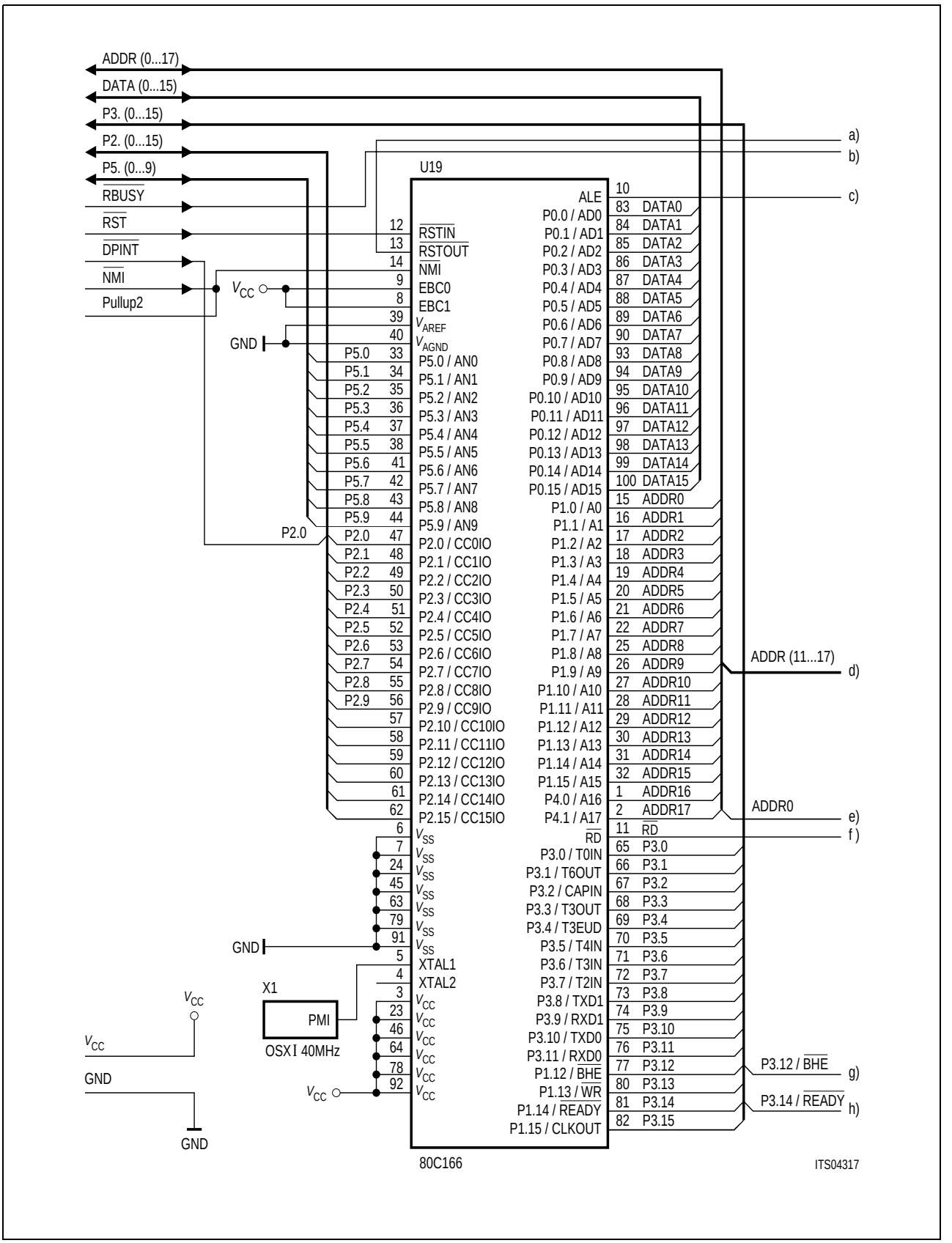
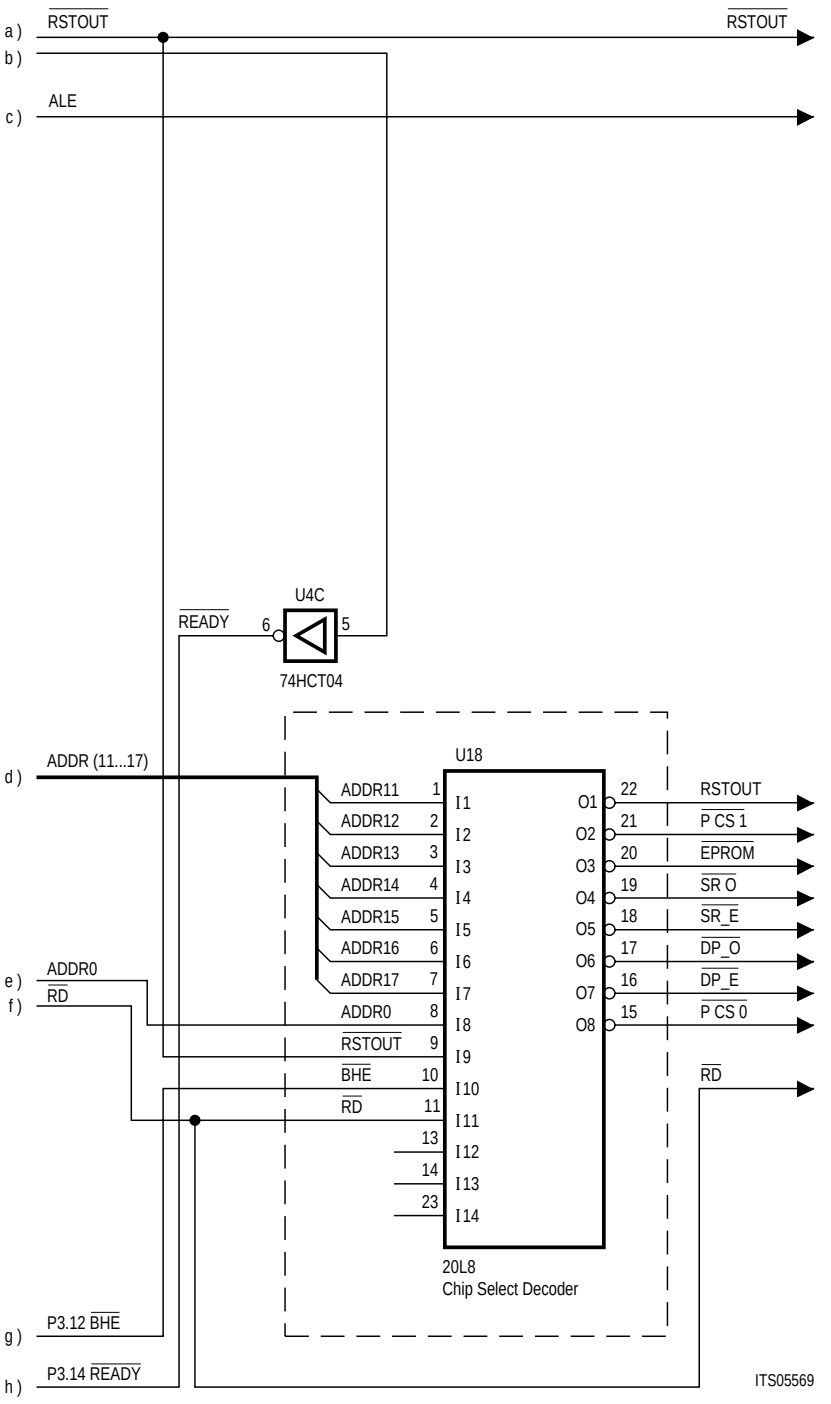


Figure 34
SAB 80C166 - CPU Block


a)
b)
c)
RSTOUT
RSTOUT
ALE
U4C
READY
6
5
74HCT04
U18
d)
ADDR (11...17)
ADDR11 1
ADDR12 2
ADDR13 3
ADDR14 4
ADDR15 5
ADDR16 6
ADDR17 7
ADDR0 8
RSTOUT 9
BHE 10
RD 11
13
14
23
I1
I2
I3
I4
I5
I6
I7
I8
I9
I10
I11
I12
I13
I14
O1 22
O2 21
O3 20
O4 19
O5 18
O6 17
O7 16
O8 15
RSTOUT
P CS 1
EPROM
SR O
SR_E
DP_O
DP_E
P CS 0
RD
20L8
Chip Select Decoder
e)
f)
ADDR0
RD
g)
P3.12 BHE
h)
P3.14 READY
ITS05569

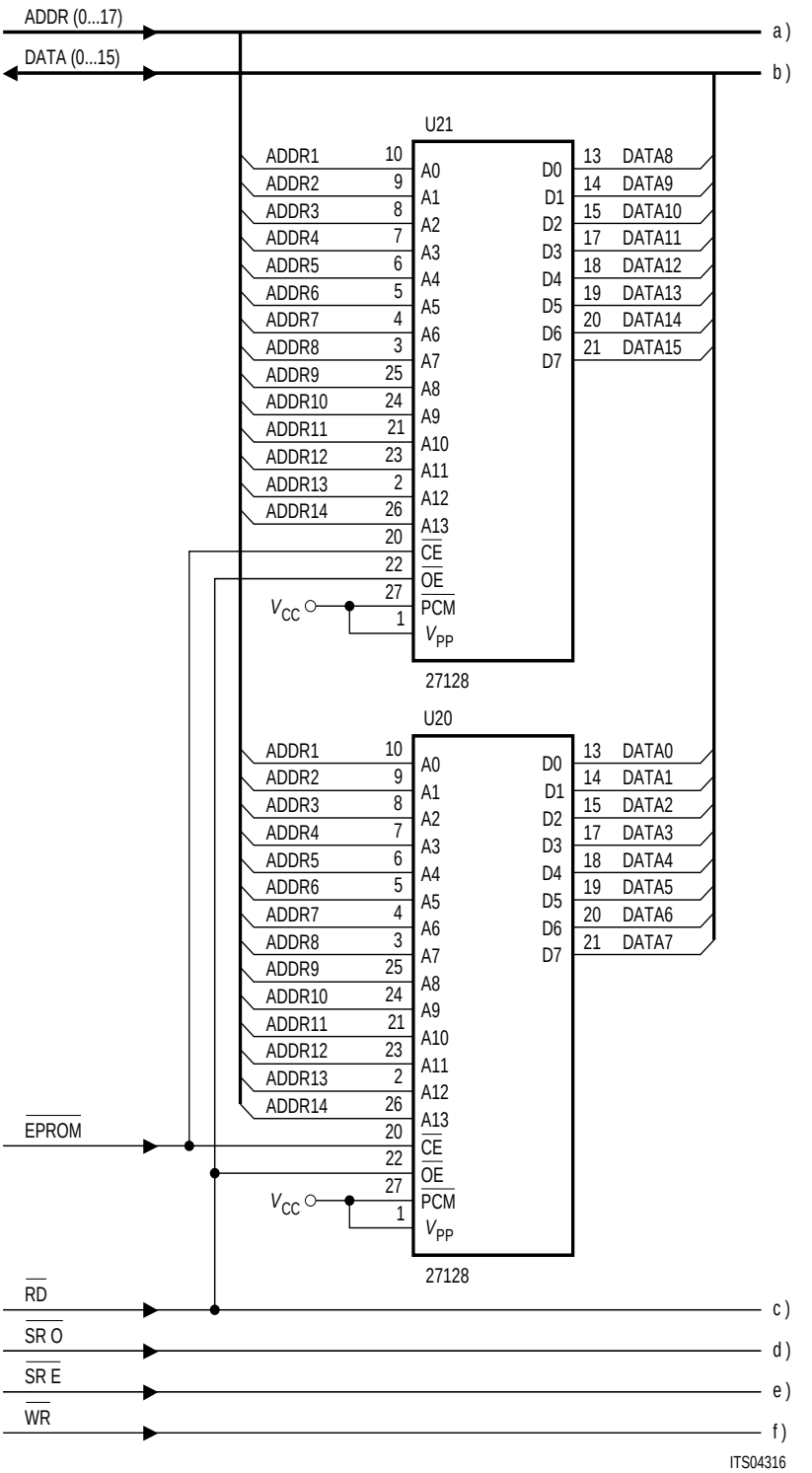
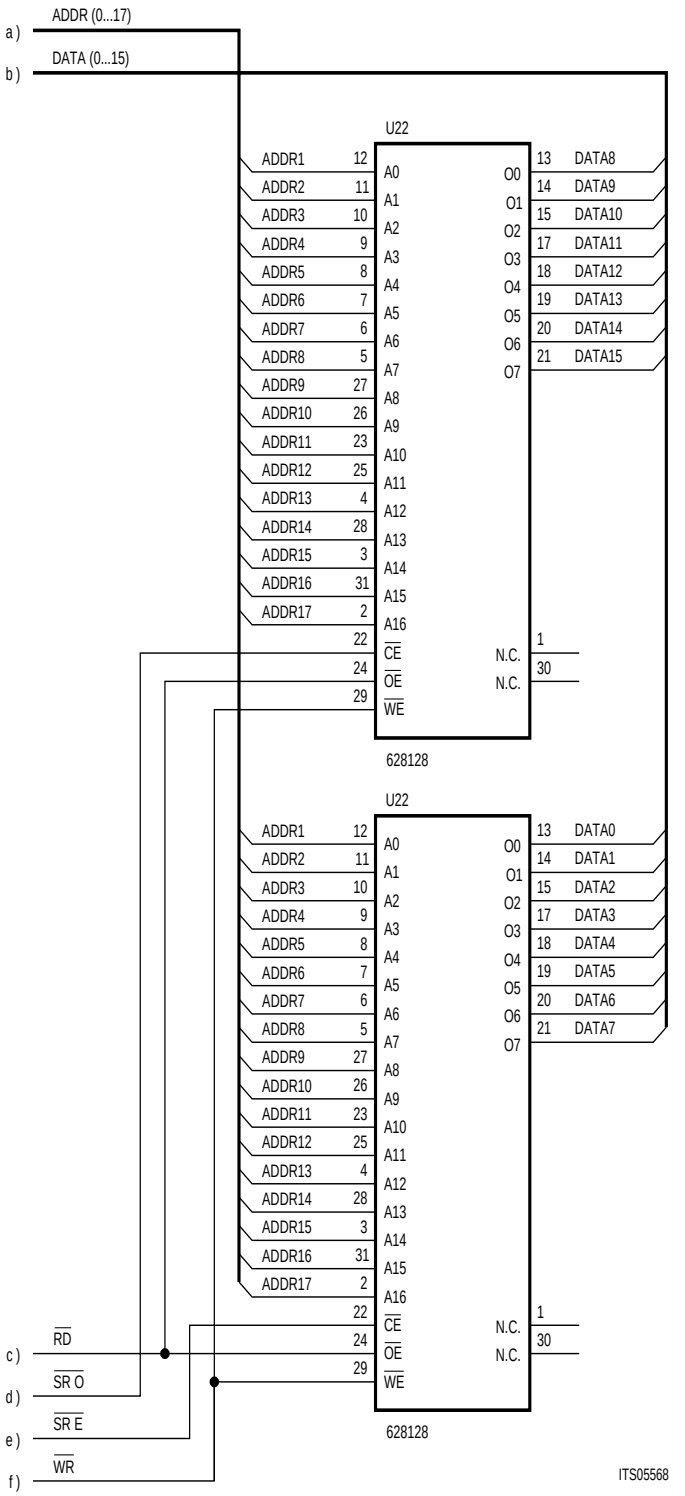


Figure 35
RAM and EPROM


a) ADDR (0...17)
b) DATA (0...15)
U22
ADDR1 12
ADDR2 11
ADDR3 10
ADDR4 9
ADDR5 8
ADDR6 7
ADDR7 6
ADDR8 5
ADDR9 27
ADDR10 26
ADDR11 23
ADDR12 25
ADDR13 4
ADDR14 28
ADDR15 3
ADDR16 31
ADDR17 2
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16
22 CE
24 OE
29 WE
O0 13 DATA8
O1 14 DATA9
O2 15 DATA10
O3 17 DATA11
O4 18 DATA12
O5 19 DATA13
O6 20 DATA14
O7 21 DATA15
N.C. 1
N.C. 30
628128
U22
ADDR1 12
ADDR2 11
ADDR3 10
ADDR4 9
ADDR5 8
ADDR6 7
ADDR7 6
ADDR8 5
ADDR9 27
ADDR10 26
ADDR11 23
ADDR12 25
ADDR13 4
ADDR14 28
ADDR15 3
ADDR16 31
ADDR17 2
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15 A16
22 CE
24 OE
29 WE
O0 13 DATA0
O1 14 DATA1
O2 15 DATA2
O3 17 DATA3
O4 18 DATA4
O5 19 DATA5
O6 20 DATA6
O7 21 DATA7
N.C. 1
N.C. 30
628128
c) RD
d) SR O
e) SR E
f) WR
ITS05568

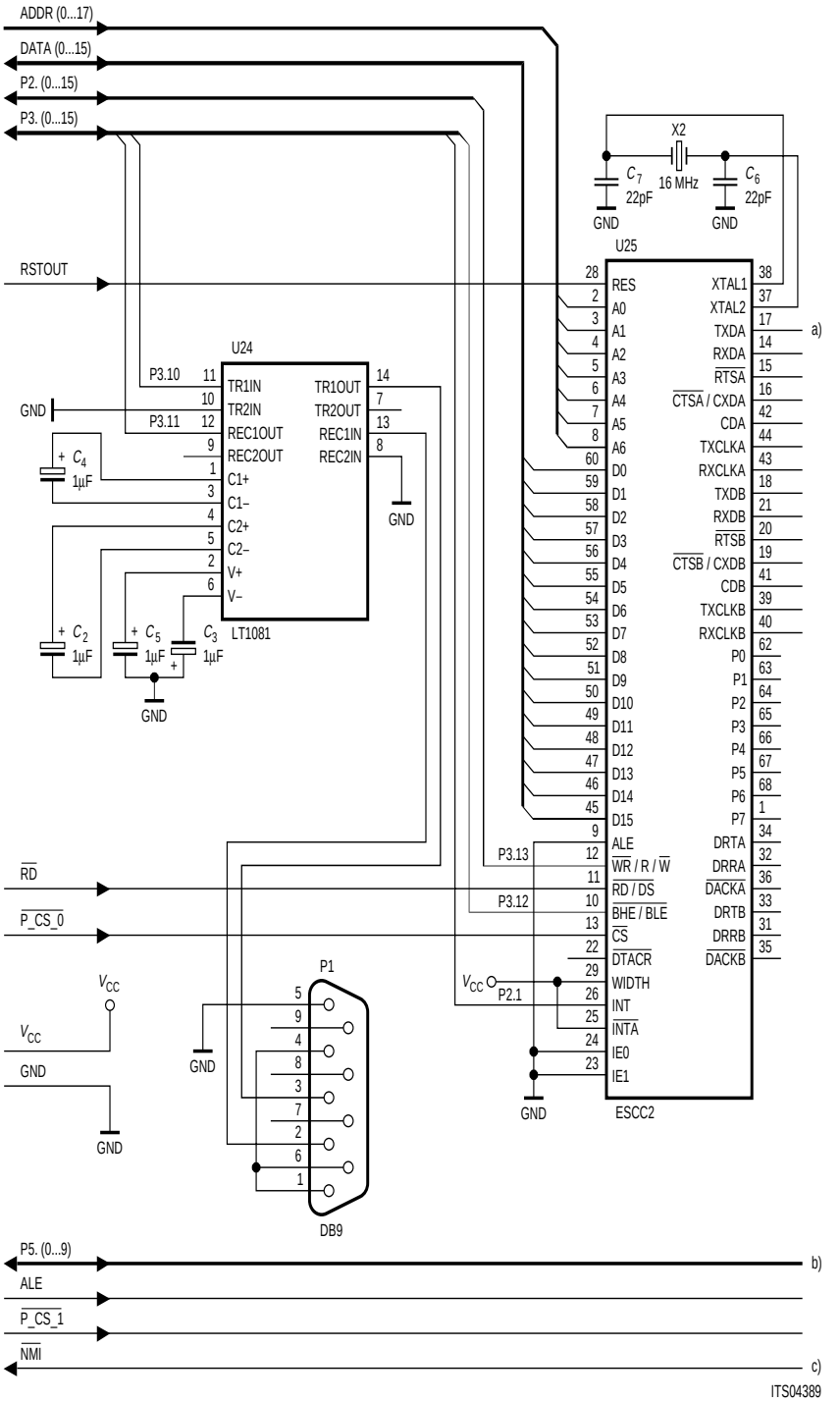
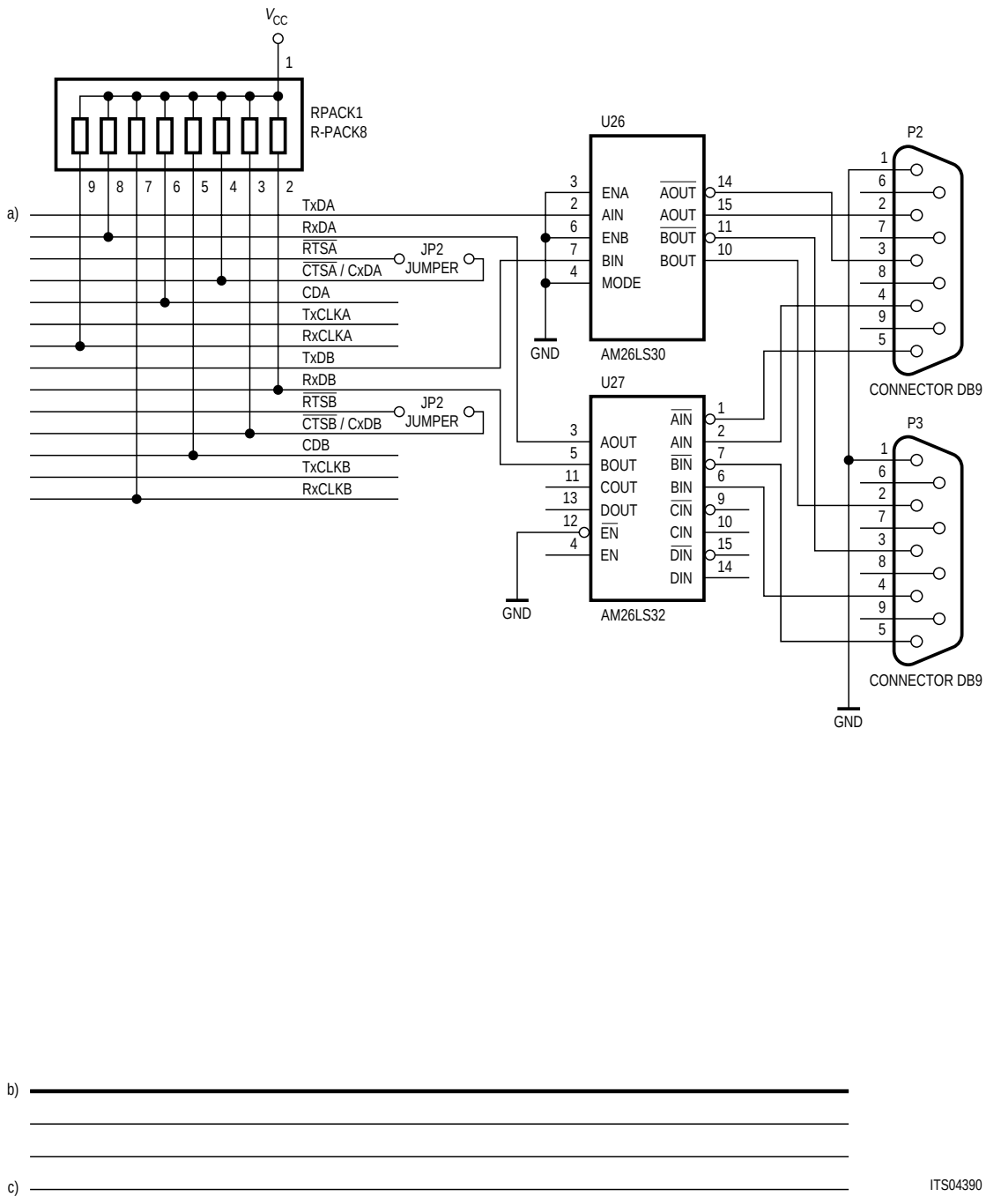


Figure 36
ESCC2 - Serial Port


V_{CC}
1
RPACK1
R-PACK8
9 8 7 6 5 4 3 2
a)
TxDA
RxDA
RTSA
CTSA / CxDA
CDA
TxCLKA
RxCLKA
TxDB
RxDB
RTSB
CTSB / CxDB
CDB
TxCLKB
RxCLKB
JP2
JUMPER
JP2
JUMPER
U26
ENA
AIN
ENB
BIN
MODE
AOUT
AOUT
BOUT
BOUT
AM26LS30
GND
U27
AOUT
BOUT
COUT
DOUT
EN
EN
AIN
AIN
BIN
BIN
CIN
CIN
DIN
DIN
AM26LS32
GND
P2
CONNECTOR DB9
P3
CONNECTOR DB9
GND
b)
c)
ITS04390

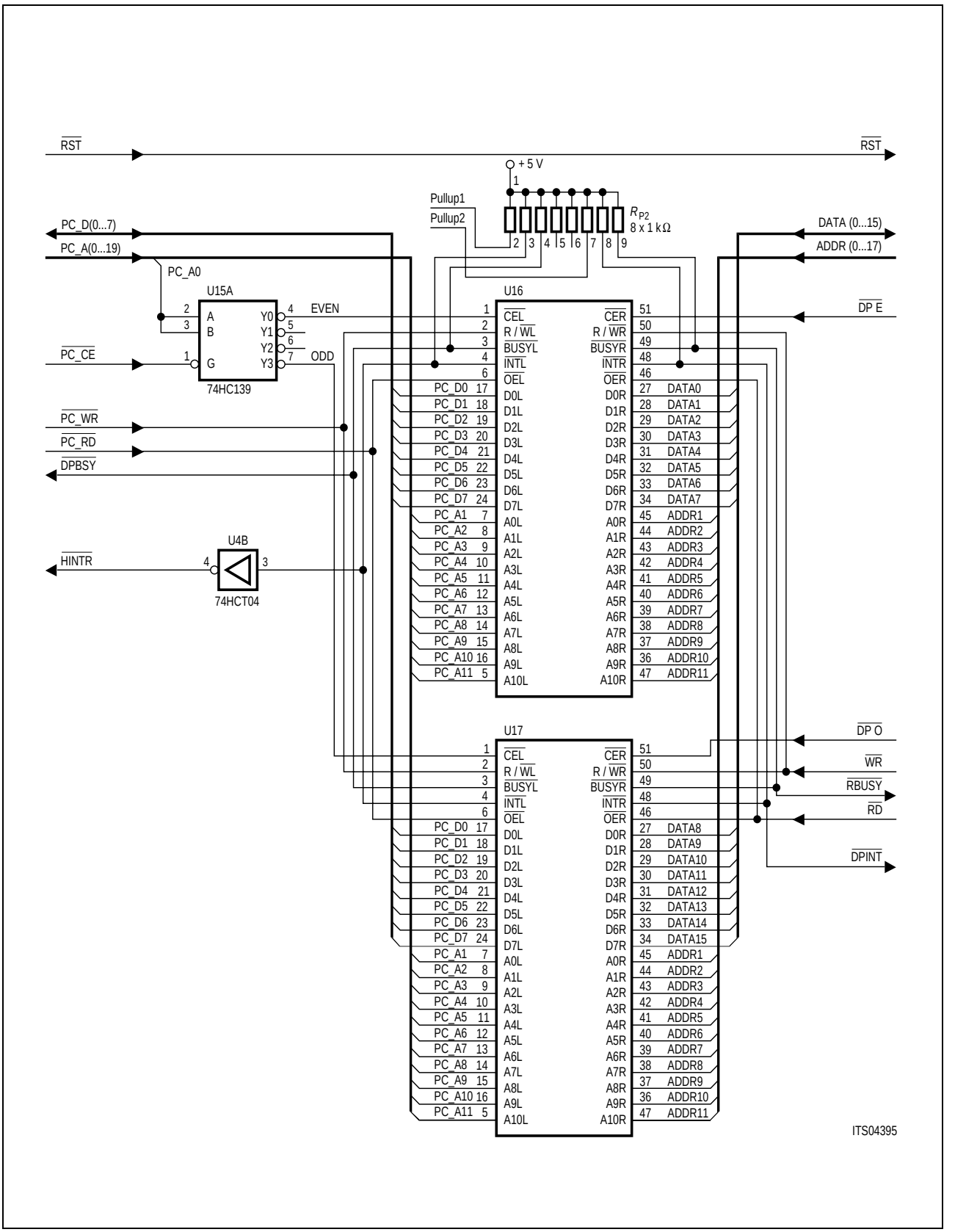


Figure 37
Communication System: ESCC2, ADMA and SAB 80C166

Appendix B

PAL-Description File

```
TITLE                DECODER_PAL1

; This decoder-PAL generates the appropriate chip select signals for EPROM, static RAM,
; Dual Port RAM and peripherals on the SAB 80C166 Evaluation Board.

CHIP                 MEM_DEC1    PALCE20V8

STRING EPROM_ADDR_SELECT      '(/A17 * /A16 * /A15)'
                              ;00000H–07FFFH
STRING DPRAM_ADDR_SELECT      '(/A17 * /A16 * A15 * A14 * A13 * /A12)'
                              ;0E000H–0EFFFH
STRING PERI_ADDR0_SELECT      '(/A17 * /A16 * A15 * A14 * A13 * A12 * /A11 * /A10)'
                              ;0F000H–0F3FFH
STRING PERI_ADDR1_SELECT      '(/A17 * /A16 * A15 * A14 * A13 * A12 * /A11 * A10)'
                              ;0F400H–0F7FFH


EQUATIONS

EPROM_CS                 = EPROM_ADDR_SELECT * RD * /RSTOUT
PERI_CS_0                = PERI_ADDR0_SELECT
PERI_CS_1                = PERI_ADDR1_SELECT
DPRAM_CS_ODD             = DPRAM_ADDR_SELECT * BHE
DPRAM_CS_EVEN            = DPRAM_ADDR_SELECT * /A0
SRAM_CS_ODD              = (/EPROM_ADDR_SELECT * (/RSTOUT + RSTOUT * /RD))
                           * /PERI_ADDR0_SELECT
                           * /PERI_ADDR1_SELECT
                           * /DPRAM_ADDR_SELECT
                           * BHE

SRAM_CS_EVEN             = (/EPROM_ADDR_SELECT * (/RSTOUT + RSTOUT * /RD))
                           * /PERI_ADDR0_SELECT
                           * /PERI_ADDR1_SELECT
                           * /DPRAM_ADDR_SELECT
                           * /A0
```

Appendix C

Source Code

Because of its volume, the source code is not attached to this description. On request, we'll send you the complete listing on floppy disk.