SIEMENS

ICs for Communications

High Speed HDLC-Frame Relay

SAB 82532 with SAB 82258 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.

As far as patents or other rights of third parties are concerned, liability is only assumed for components, not for applications, processes and circuits implemented within components or assemblies.

The information describes the type of component and shall not be considered as assured characteristics.

Terms of delivery and rights to change design reserved.

For questions on technology, delivery and prices please contact the Semiconductor Group Offices in Germany or the Siemens Companies and Representatives worldwide (see address list).

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Siemens Office, Semiconductor Group.

Siemens AG is an approved CECC manufacturer.

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**

1 General Information

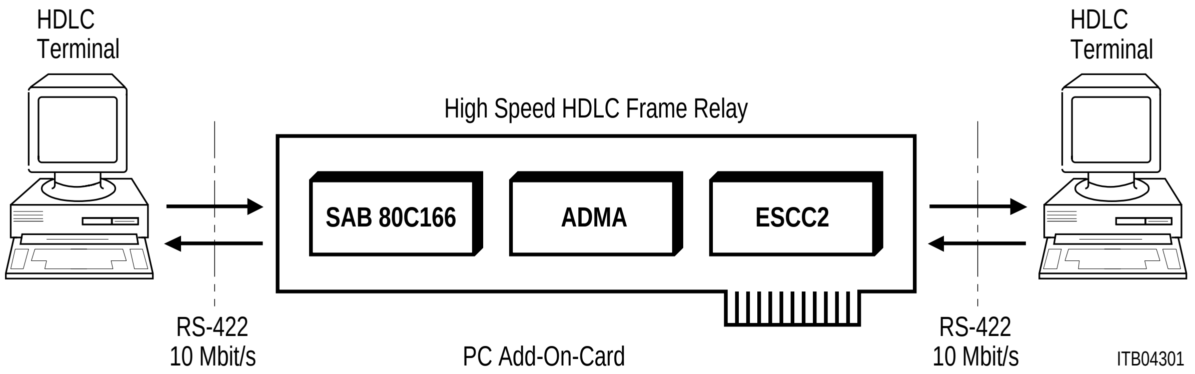
SAB 82532 with SAB 82258 and SAB 80C166

High Speed HDLC-Frame Relay

This application note describes a simple application example for an high-speed HDLC-frame relay using the Enhanced Serial Communication Controller ESCC2 (SAB 82532), the Advanced DMA-Controller ADMA (SAB 82258) and the 16-Bit Microcontroller SAB 80C166. Transfer rates up to 10 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 82258 and the SAB 80C166 for high speed communication purposes.

Contents

- Chapter 1
  General Overview
- Chapter 2
  How to integrate the ESCC2 with ADMA into an 80C166 microprocessor system. All hardware aspects concerning the ESCC2, ADMA and 80C166 are explained.
- Chapter 3
  Detailed description of how the ESCC2 can be programmed to handle HDLC-communication for higher data rates supported by the DMA-controller ADMA. The basic architecture of an object oriented Device Driver Module for the ESCC2 is explained. Furthermore an idea of the cooperation between the ADMA-device driver module and the ESCC2-device driver module is given. The initialization and the programming of the ADMA is described.



Functional Block Diagram

- Chapter 4
  A very simple application program module for the HDLC-frame relay on top of the ESCC2-device driver software and the underlying run-time software is described.
- Chapter 5
  The performance of the system thus realized is analyzed and presented. Especially the aspects of frame length, frame rate and the operational overhead are discussed.
- Chapter 6
  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 relay software.
- 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 implement the software by using the software development tools from Tasking.

Remarks

It is recommended to have a basic understanding of the ESCC2-interrupt interface as well as of the ESCC2 DMA-interface described in the ESCC2 Technical Manual. For details concerning the ADMA or 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 on to 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 ADMA and the 80C166 for communication subsystems in general. Therefore the complexity of the software environment has been reduced to a minimum.

2 Introduction

This application note describes a design example for a High Speed HDLC Frame Relay using the SAB 82532 (Enhanced Serial Communication Controller – 2 Channels, ESCC2), the SAB 82258 (Advanced DMA-Controller, ADMA) and the microcontroller SAB 80C166. It is recommended that before reading this application note, the user has a basic understanding of the interrupt and DMA-interface of the SAB 82532, described in the “ESCC2 Technical Manual”.

In this description of the high speed HDLC-frame relay two basic subjects are discussed.

The first one is how the SAB 82532 with the DMA-controller ADMA can be integrated into a hardware environment based on a 16-bit microcontroller from Siemens, the SAB 80C166. The SAB 82532 as a peripheral device can be connected to the external bus of the SAB 80C166 without using additional glue logic in the form of external hardware. The integration of the DMA-controller SAB 82258, however, requires hardware components in the form of a bus controller, transceivers and so forth since the coprocessor ADMA is able to operate as a bus master.

The second subject is the software of the high speed HDLC-frame relay. The special device drivers for the ESCC2 and the ADMA have the advantage of utilizing the devices’ features concerning DMA-transfer and interrupt treatment to enhance performance. The tasks of data transfer and interrupt servicing are shared effectively between the DMA-controller and the microcontroller.

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

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

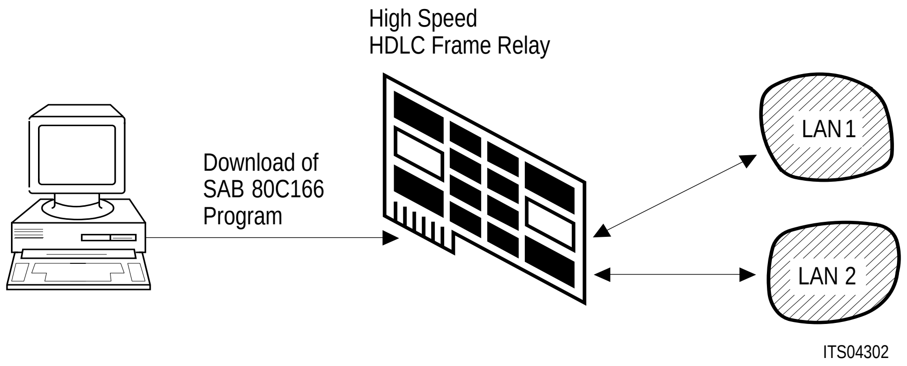


Figure 1
Application HDLC-Frame Relay

3 Hardware of the High Speed HDLC Frame Relay

The hardware of the high speed HDLC-frame relay is realized as a plug-in PC-board. Beside the microcontroller SAB 80C166 and the DMA-controller ADMA important elements of the hardware are listed in the following: a 64 K EPROM containing the firmware, a 256 K RAM, a bus controller, address latches 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 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-frame relay. The equations to decode the corresponding chip select signals are listed in Appendix B.

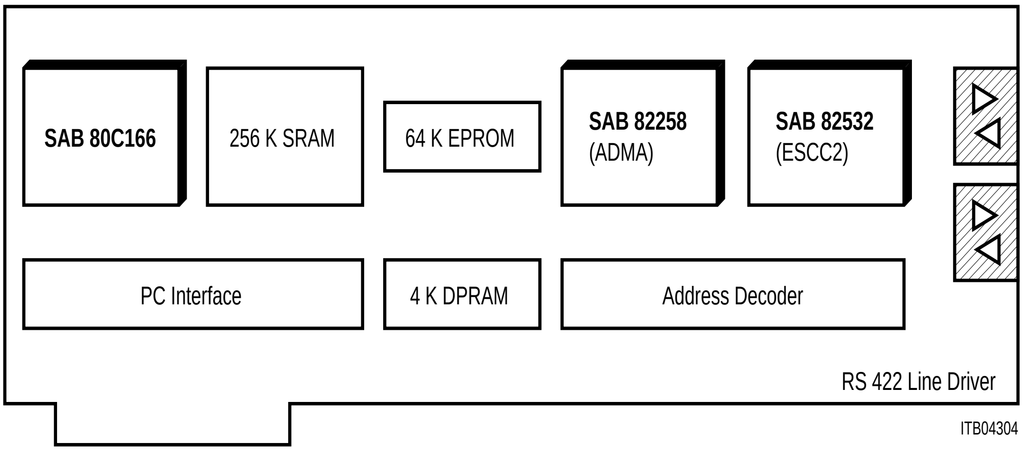


Figure 2
Functional Block Diagram

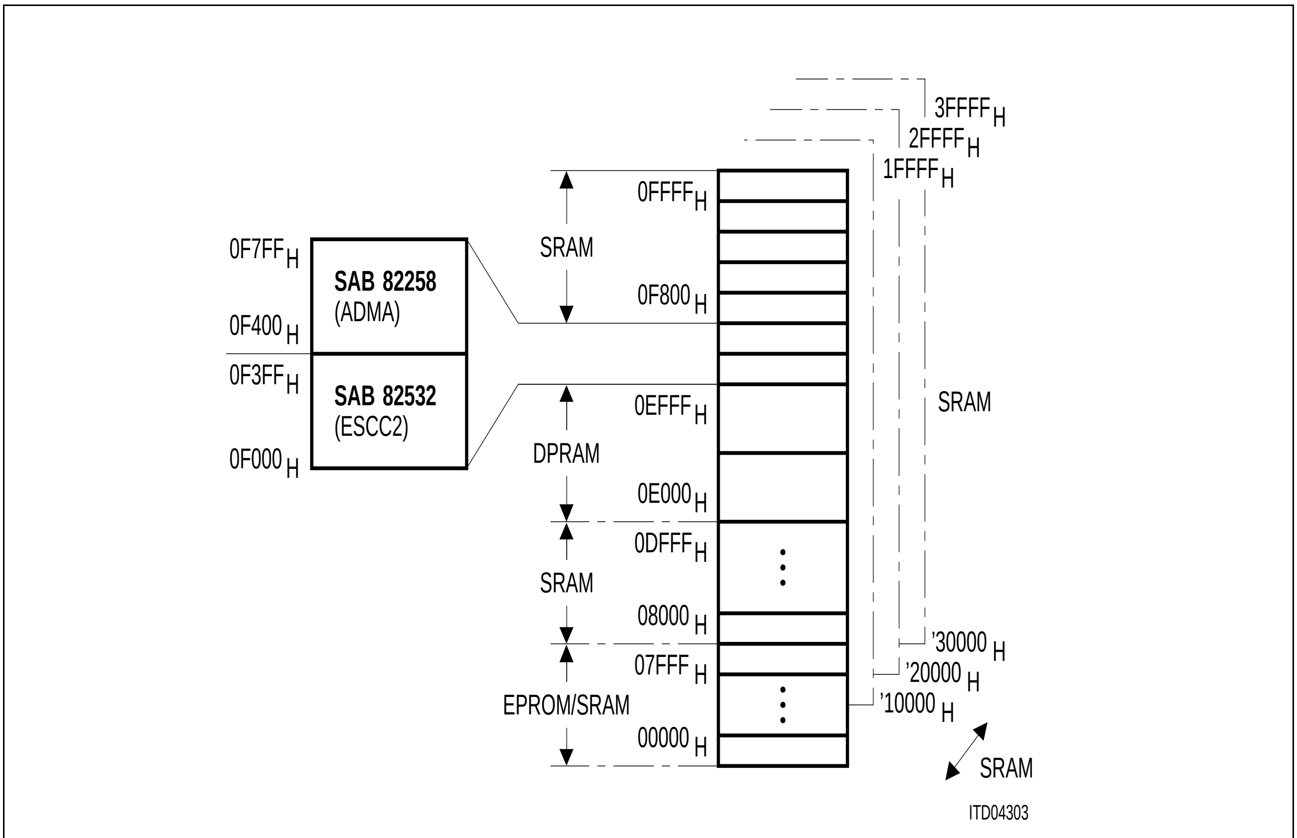


Figure 3
Memory Organization

3.1 ESCC2 with SAB 80C166

Microprocessor Interface

Practically no external components are required to adapt the ESCC2 to the SAB 80C166. The ESCC2 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.

Interrupts

Since no interrupt acknowledge cycle is supported by SAB 80C166 pin $\overline{INTA}$ is deactivated by connecting $\overline{INTA}$ to + 5 V. Because the application includes only one ESCC2, 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 to mask interrupts for a short time before leaving the interrupt function. In this way a new edge can be forced, because the SAB 82532 stores pending interrupts and reactivates the interrupt line when the interrupt mask is withdrawn.

Reset

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

Serial Interface

The input signals RxDn, 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 $\overline{\text{RTSn}}$ and $\overline{\text{CTSn}}$ are connected to each other.

The physical layer is realized with an 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 ESCC2.

Timing Characteristics

The timing of the microcontroller can be adapted to that of the ESCC2 by changing via 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. The programmed read/write delay causes a delay by a quarter of a machine cycle (25 ns at f_{OSC} = 40 MHz). This delay does not by itself extend the memory cycle time, because through this programmed delay 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 to or longer than 10 ns (SAB 80C166: 0 ns). The hold time has to be lengthened with an additional memory tristate time wait state.

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 one selected address space. In section 2.2 an example is given for programming the bus configuration register matching the timing specification of the ESCC2 and the ADMA.

3.2 ADMA with SAB 80C166

In this application the ADMA operates as a coprocessor on the same bus as the microcontroller. This operating mode is called local mode. **Figure 4** shows the architecture of the system. The bus management is controlled by a HOLD/HLDA-handshake.

Microprocessor Interface

The ADMA has an adaptive bus interface; so the bus is configured in the non multiplex ’286 mode of the SAB 80C166. When ADMA is bus master, it generates a write/read request by activating the bus status lines $\overline{\text{S0}}$ or $\overline{\text{S1}}$: therefore a bus controller has to be added which generates the corresponding commands $\overline{\text{WR}}$ and $\overline{\text{RD}}$ from $\overline{\text{S0}}$ and $\overline{\text{S1}}$.

Since the ADMA uses the pipelined addressing mode the addresses of the following bus cycle are valid although the current bus cycle is not finished yet. For that reason external address latches have to buffer the addresses during the whole bus cycle. The address latches are controlled by the bus controller, too.

Assuming that the SAB 80C166 is bus master the internal registers of the ADMA have to be accessible. Hence the bidirectional, lower eight address lines of the ADMA are driven by transceivers.

The transceivers are activated if the SAB 80C166 is bus master; the latches are activated if the ADMA is bus master. In this way a conflict on the address bus is avoided.

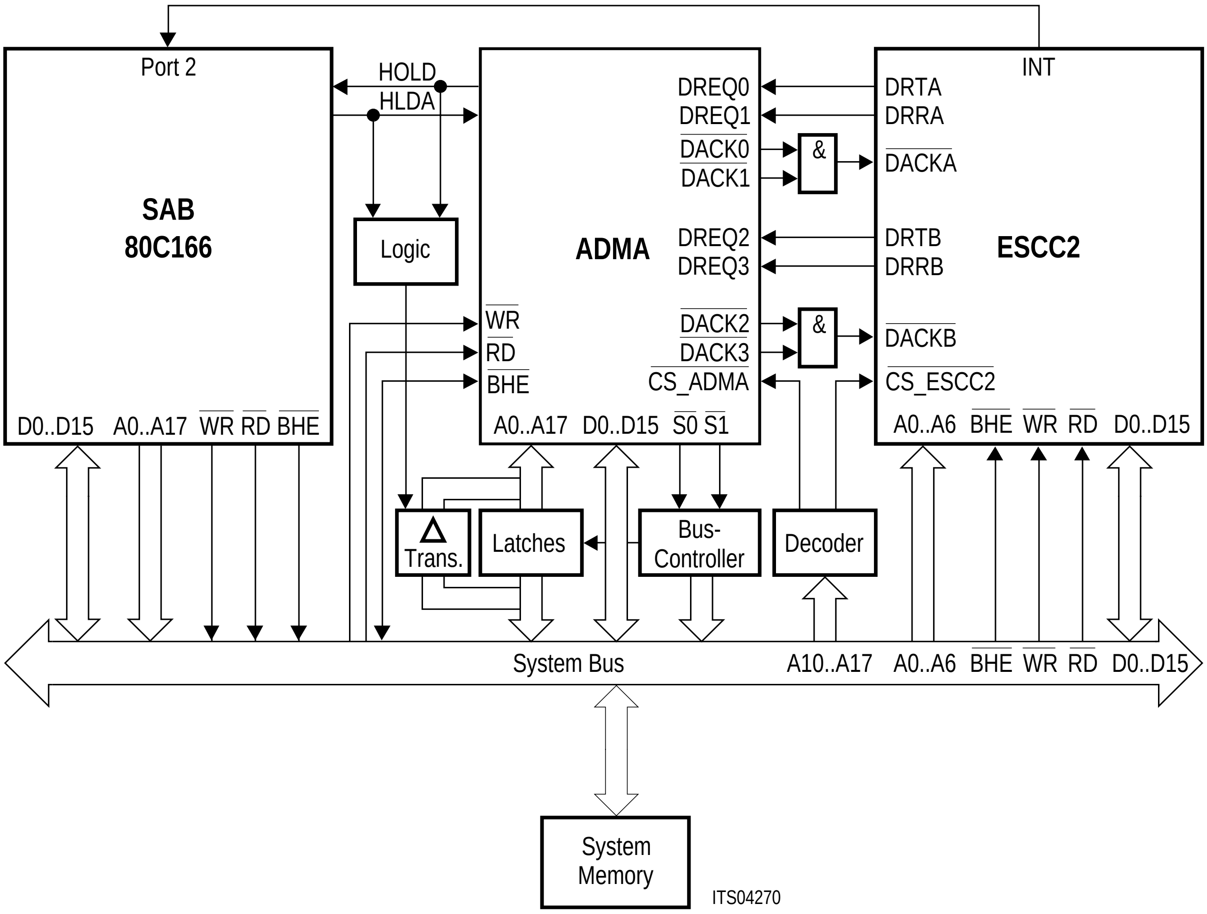


Figure 4
Architecture of System

To avoid a conflict on the control lines the signals $\overline{\text{WR}}$ and $\overline{\text{RD}}$ generated by the bus controller are also driven by transceivers. These transceivers are activated if the ADMA is bus master.

Reset

When activating the reset input, the ADMA is forced into its initial state. While the reset input is active, line A23 must be forced to the appropriate level to select the desired bus interface mode (286 mode).

Bus Arbitration

To arbitrate access to the bus between the SAB 82258 and the microcontroller, the signals HOLD and HLDA serve for communication. Normally the ADMA competes for the bus via HOLD, the microcontroller grants access to the bus via HLDA. The HLDA-signal can also be deactivated in order to force the ADMA off the bus for a certain reason (kick off). The bus arbitration is controlled by a HOLD/HLDA-handshake.

In order to support multi-master systems and communication with external DMA-functions, a bus arbitration feature is implemented in the SAB 80C166. The signals $\overline{\text{HOLD}}$ and $\overline{\text{HLDA}}$ are implemented as second alternate functions at pins P2.15 ($\overline{\text{HOLD}}$) and P2.14 ($\overline{\text{HLDA}}$). The control bit HLDEN of the Processor Status Word register (PSW) has to be programmed to ‘1’ to enable the bus arbitration function of these pins.

Slave Interface

The slave interface is used to access the ADMA’s internal registers. Although nearly all of the communication between CPU and ADMA is done via memory based data blocks, some direct accesses to the ADMA-registers are necessary.

For example during the initialization phase the General Mode Register must be written, or to start a channel the Command Pointer Register and the General Command Register must be loaded.

The slave interface is enabled by the $\overline{\text{CS}}$-input and consists of the following lines:

- $\overline{\text{RD}}$, $\overline{\text{WR}}$ – control lines (input)
- A0 – A7 – register address (inputs)
- D0 – D15 – data lines (inputs/outputs)

Since the microcontroller and the DMA-controller have different clocks asynchronous access by using the control lines $\overline{\text{RD}}$ and $\overline{\text{WR}}$ is used.

Timing Characteristics

Similar to the ESCC2 the timing of the microcontroller can be adapted to that of the ADMA by changing via software the default values of the SAB 80C166 (refer to chapter 2.1).

To guarantee the asynchronous access setup time a fast PAL and a fast address transceiver have to be used. Moreover a read write delay has to be programmed.

Additionally the $\overline{\text{RD}}$-impulse width has to be lengthened by five wait states in order to match the timing specification of the ADMA.

The modifications described here and them concerning the ESCC2 can be programmed in the bus configuration register using the _bfld (bit field) instruction:

```
_bfld (BUSCON1, 0x00FF, 0x00CA);
```

The _bfld instruction assigns the constant 0x00CA to the bit field indicated by the constant mask 0x00FF of the bitaddressable operand BUSCON1. This statement configures the bus interface with one read/write delay, five wait states and one memory tristate time wait state. For more information e.g. address range selection refer to listings of the DDSH-module in Appendix C (DdshInit ()).

3.3 ESCC2 with ADMA

DMA-Interface

The DMA-interface consists of three lines:

●	DREQ	(– DMA-request)
●	$\overline{\text{DACK}}$	(– DMA-acknowledge)
●	$\overline{\text{EOD}}$	(– End of DMA)

The first two lines work as request and acknowledge lines to control synchronized DMA-transfers as known from conventional DMA-controllers.

The $\overline{\text{EOD}}$-signals are not used in the application at hand.

Transfer Modes

Each channel controls data transfer in two basic operating modes:

- single-cycle mode
- two-cycle mode

In single-cycle mode, bytes or words are transferred directly from the data source to the data destination in a single bus cycle per transfer. Using SAB 82C258-A 12 with 25 MHz this mode enables a total data rate of up to 12.5 Mbytes per second, and that as single channel data rate or as a cumulative data rate of multiple channels. Thus the advantage in single-cycle mode lies in speed.

In two-cycle mode, source data is always stored in the ADMA before being sent of the destination. Although half as fast as single-cycle transfer, it has several compensating advantages. Since the data actually enters the controller, there is a possibility to act on the data.

As a very useful feature for single-cycle transfers, the ESCC2 supports the optional inversion of the functions of read/write control lines. If programmed via register CCR2 (RWX bit is set) $\overline{\text{RD}}$ and $\overline{\text{WR}}$ are exchanged internally in Intel bus interface mode while $\overline{\text{DACK}}$ is active **(see figure 5)**.

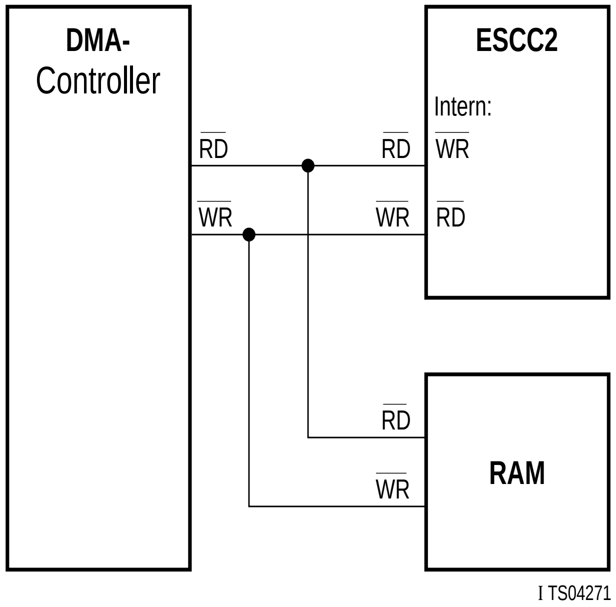


Figure 5
$\overline{RD}$-/$\overline{WR}$-Exchange

The application at hand operates in two-cycle mode. Using single-cycle mode the following problem arises:

According to the ESCC2-specification the DRR is reset with the falling edge of $\overline{RD}$ during the last read access to RFIFO. The $\overline{RD}$-signal is generated by the bus controller from $\overline{S1}$ activated by the DMA-controller. The ADMA however checks the DREQ-signal in view of the next DMA-cycle before the bus controller has generated the $\overline{RD}$-signal from the corresponding $\overline{S1}$. That means that the trailing edge of DREQ is received later, a continuous request is assumed and subsequent transfers will be executed. There is no hardware workaround to solve this problem. In chapter 4 a software workaround is described.

Timing Characteristics

To guarantee correct operation for SAB 82C258-A 12 with 25 MHz a fast PAL and a fast control line ($\overline{RD}$ and $\overline{WR}$) transceiver have to be used.

4 Device Driver Modules

4.1 Overview

To run the application the following tasks have to be executed:

- the device driver system has to be initialized
- the devices have to be initialized
- messages have to be transferred from the application module to the ESCC2-device driver module and vice versa
- the channel programs of the ADMA have to be prepared; the ADMA-channels have to be started
- the channel programs have to be fetched from RAM
- interrupts have to be served
- data have to be transferred from RAM to the ESCC2 and vice versa

These tasks are executed either by the SAB 80C166 or by the DMA-controller ADMA. **Figures 6** to **9** give an overview of the specific tasks in sequence. The upper corner at the right side of each element contains the device that executes the corresponding task. **Figure 6** shows the sequence of initialization; after that data can be received or transmitted. The procedure of transmission is controlled by transmit requests of the application as well as by DMA-requests and XPR-interrupts of the ESCC2 (**figures 7/8**). The procedure of receiving is controlled by DMA-requests and RME-interrupts of the ESCC2 (**figure 9**). In the following an example is given to design appropriate software.

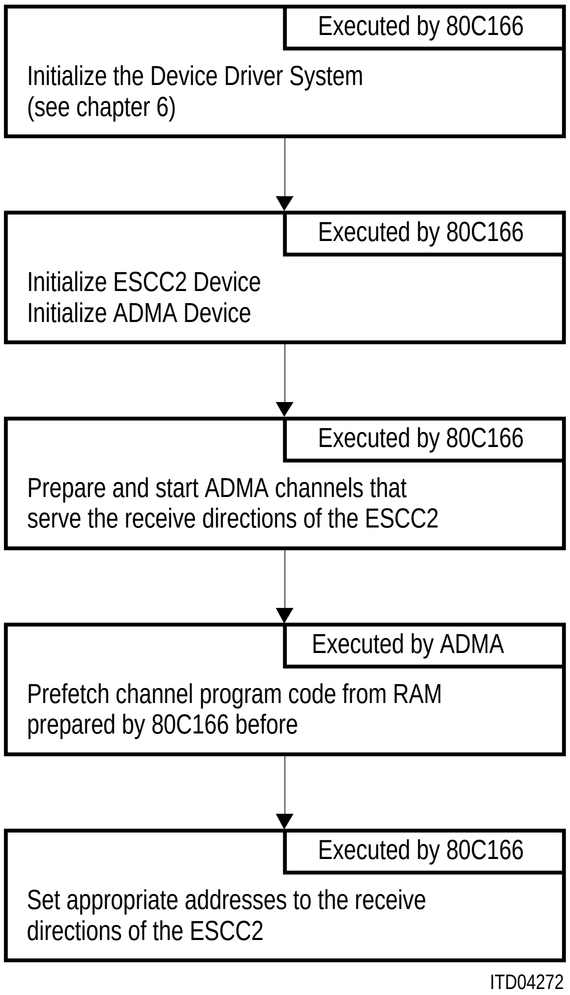

Executed by 80C166
Initialize the Device Driver System
(see chapter 6)
Executed by 80C166
Initialize ESCC2 Device
Initialize ADMA Device
Executed by 80C166
Prepare and start ADMA channels that
serve the receive directions of the ESCC2
Executed by ADMA
Prefetch channel program code from RAM
prepared by 80C166 before
Executed by 80C166
Set appropriate addresses to the receive
directions of the ESCC2
ITD04272

Figure 6
Sequence of Initialization Actions

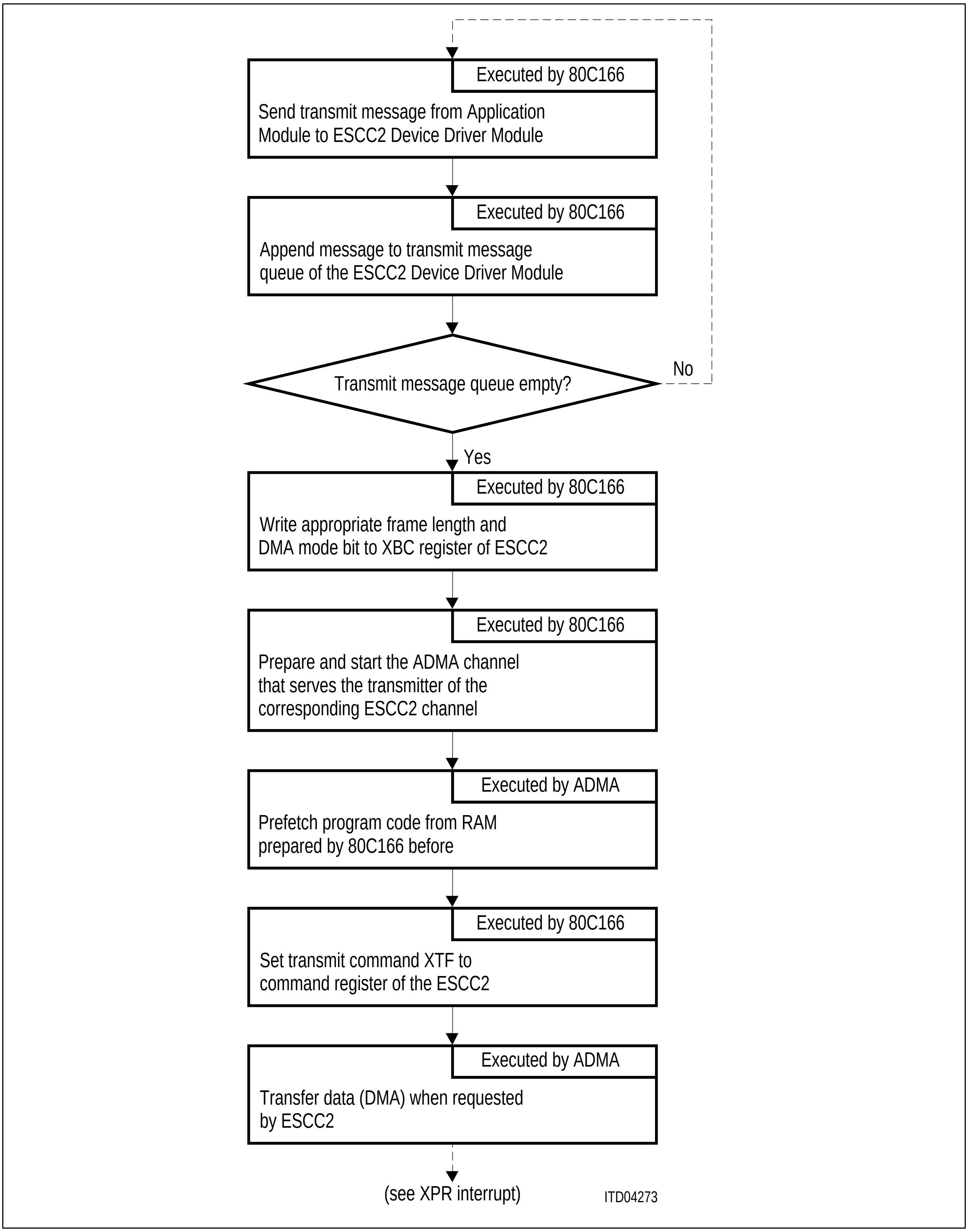

Executed by 80C166
Send transmit message from Application Module to ESCC2 Device Driver Module
Executed by 80C166
Append message to transmit message queue of the ESCC2 Device Driver Module
Transmit message queue empty?
No
Yes
Executed by 80C166
Write appropriate frame length and DMA mode bit to XBC register of ESCC2
Executed by 80C166
Prepare and start the ADMA channel that serves the transmitter of the corresponding ESCC2 channel
Executed by ADMA
Prefetch program code from RAM prepared by 80C166 before
Executed by 80C166
Set transmit command XTF to command register of the ESCC2
Executed by ADMA
Transfer data (DMA) when requested by ESCC2
(see XPR interrupt)
ITD04273

Figure 7
Sequence of Transmitter Actions after Transmit Request by Application

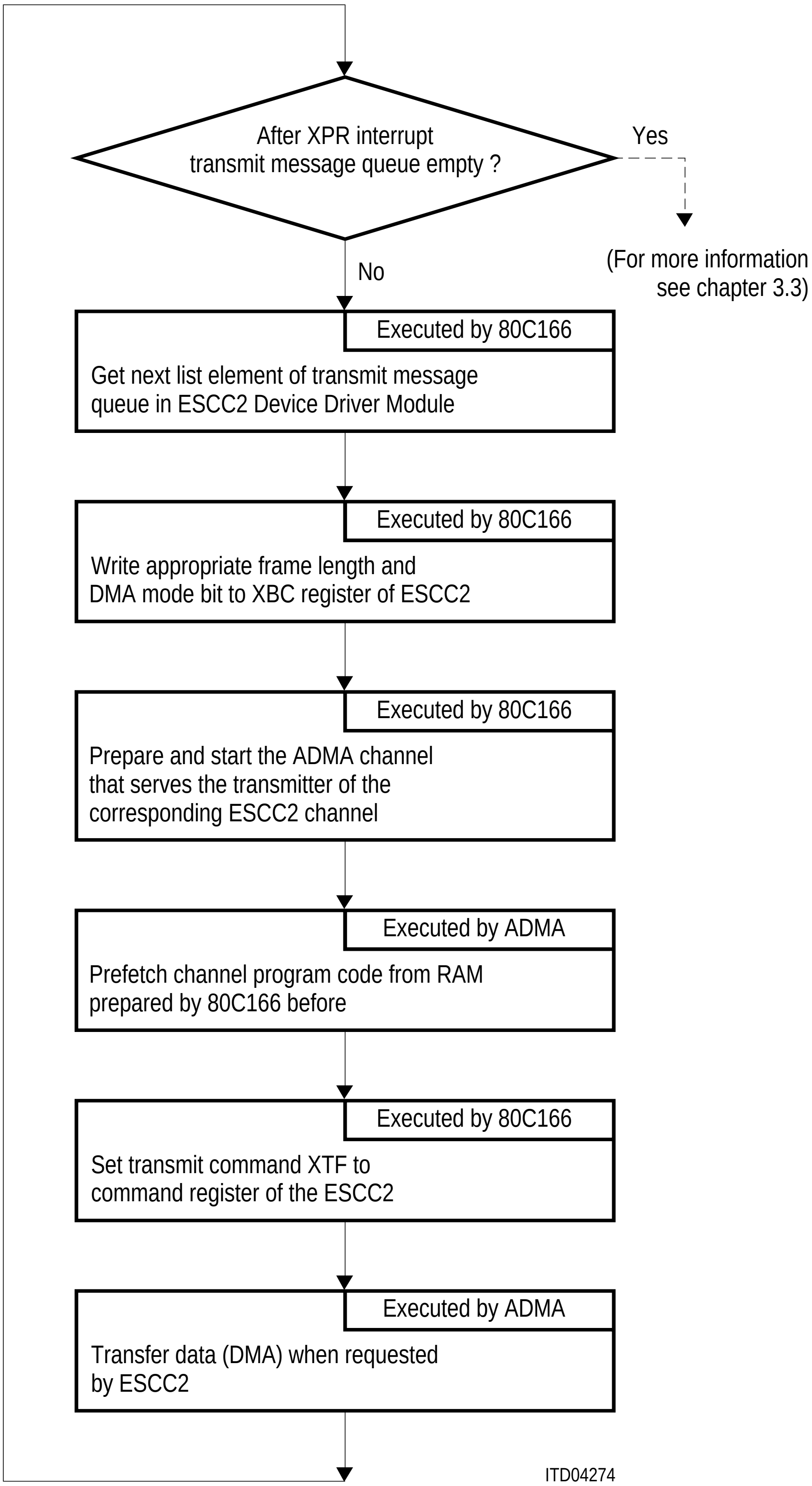

After XPR interrupt
transmit message queue empty ?
Yes
(For more information
see chapter 3.3)
No
Executed by 80C166
Get next list element of transmit message
queue in ESCC2 Device Driver Module
Executed by 80C166
Write appropriate frame length and
DMA mode bit to XBC register of ESCC2
Executed by 80C166
Prepare and start the ADMA channel
that serves the transmitter of the
corresponding ESCC2 channel
Executed by ADMA
Prefetch channel program code from RAM
prepared by 80C166 before
Executed by 80C166
Set transmit command XTF to
command register of the ESCC2
Executed by ADMA
Transfer data (DMA) when requested
by ESCC2
ITD04274

Figure 8
Sequence of Transmitter Actions after XPR-Interrupt

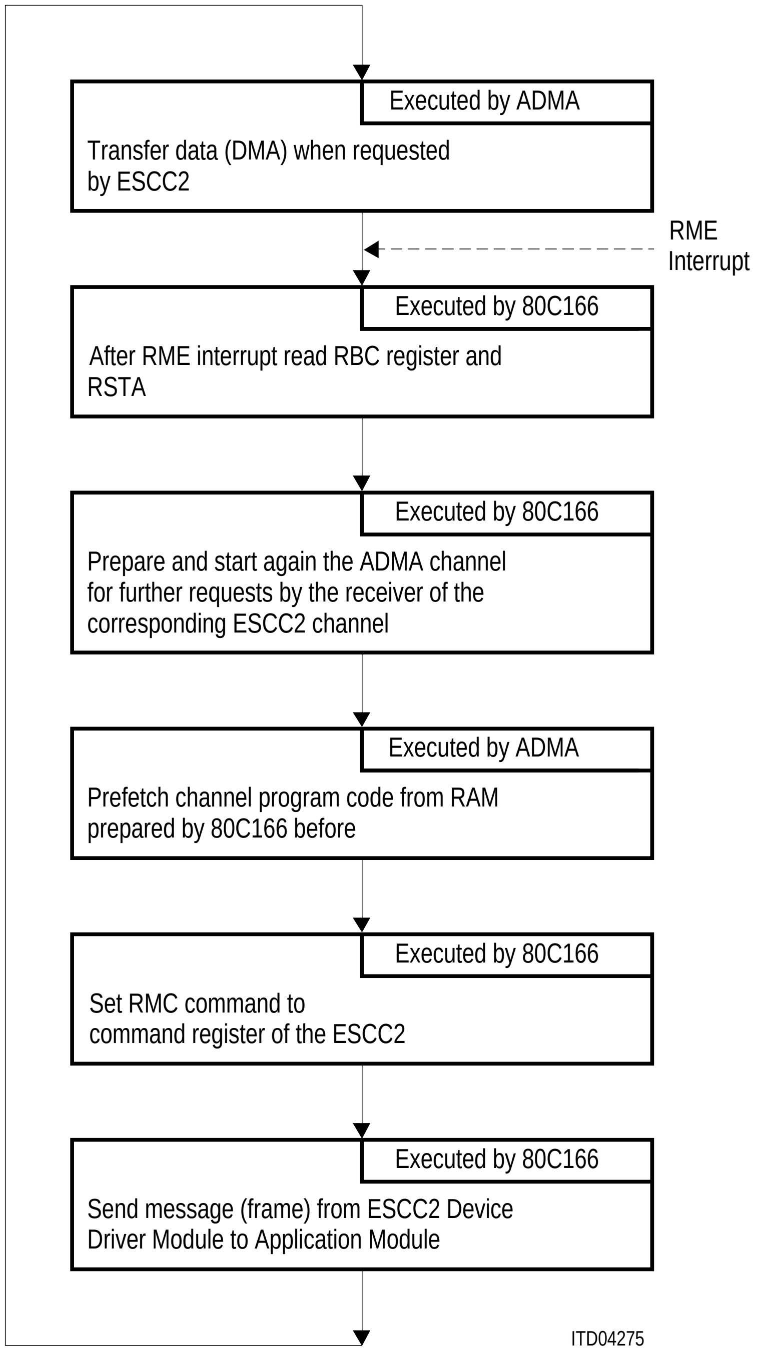

Executed by ADMA
Transfer data (DMA) when requested by ESCC2
RME Interrupt
Executed by 80C166
After RME interrupt read RBC register and RSTA
Executed by 80C166
Prepare and start again the ADMA channel for further requests by the receiver of the corresponding ESCC2 channel
Executed by ADMA
Prefetch channel program code from RAM prepared by 80C166 before
Executed by 80C166
Set RMC command to command register of the ESCC2
Executed by 80C166
Send message (frame) from ESCC2 Device Driver Module to Application Module
ITD04275

Figure 9
Sequence of Receiver Actions

The device driver modules ESCC2 and ADMA, integrated into the device driver system's environment, represent the connection to the Siemens devices (Enhanced Serial Communication Controller, ESCC2 and Advanced DMA-Controller, ADMA). Therefore, referring to the programming of the devices by the user, these modules have to be understood as the interface modules between the application level and the devices themselves. The functional position of the device driver modules in the system is shown in **figure 10**.

The ESCC2-device driver is able to send and to receive messages via the device driver system and to serve interrupts generated by the ESCC2-device itself. The ADMA-device driver module provides nine service routines to control the ADMA-device and the data transfer. **Figure 11** shows the correspondence between the serial channels of the ESCC2 and the DMA-channels of the ADMA; in addition the service routines provided by the ADMA-module are listed.

Supported by the device driver system the application program module HSHFR is able to direct individual tasks to the ESCC2-device driver module. The ESCC2-device driver module receives these messages and passes them to the ESCC2 and – via service routines of the ADMA-device driver module – to the ADMA, matching the device's programming specification.

The initialization and the control of the devices are performed by the CPU. The data are transferred by the ADMA, when a DMA-transfer request occurs.

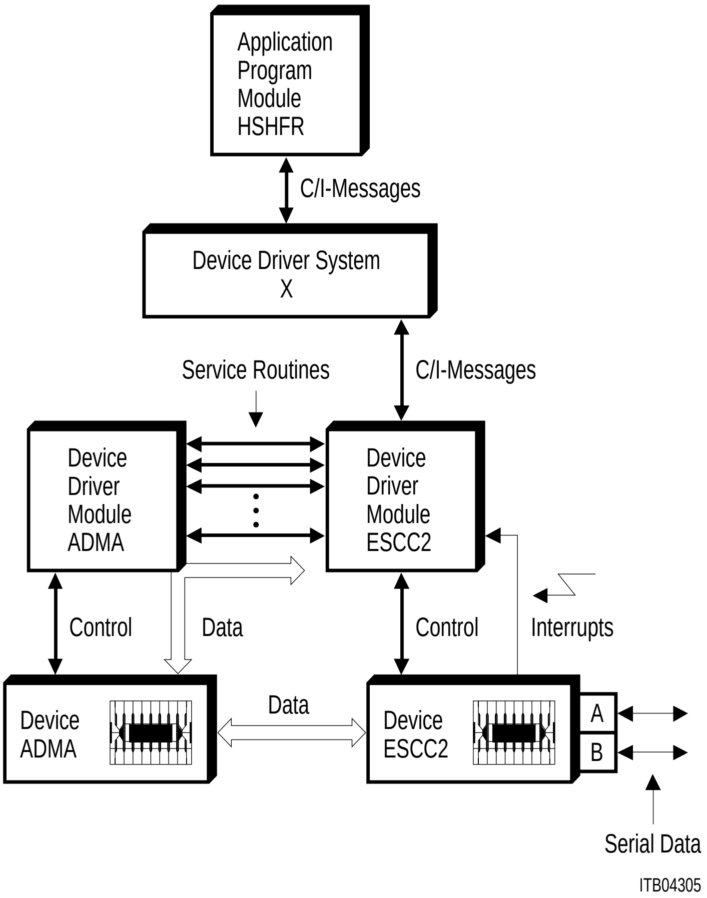


Figure 10
Device Driver Modules

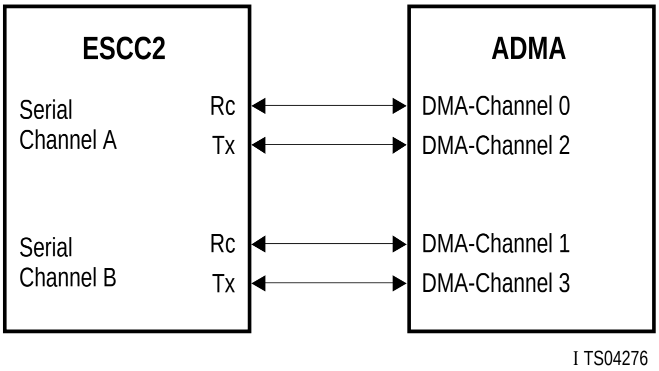


Figure 11
Correspondence between Serial Channels of ESCC2 and DMA-Channels of ADMA

Service Routines

InitAdmaDevice

InitAdmaChanRc [ESCC2_CH_A] = InitAdmaChan0
InitAdmaChanRc [ESCC2_CH_B] = InitAdmaChan1
InitAdmaChanTx [ESCC2_CH_A] = InitAdmaChan2
InitAdmaChanTx [ESCC2_CH_B] = InitAdmaChan3

StartAdmaChanRc [ESCC2_CH_A] = StartAdmaChan0
StartAdmaChanRc [ESCC2_CH_B] = StartAdmaChan1
StartAdmaChanTx [ESCC2_CH_A] = StartAdmaChan2
StartAdmaChanTx [ESCC2_CH_B] = StartAdmaChan3

Before that the CPU has to

- generate a channel specific “command block” somewhere in memory,
- inform the ADMA where that block is and
- give a “start channel” command.

An important difference between interrupt mode and DMA-mode is that using DMA-mode the device driver module ESCC2 hasn’t to distinguish between short frames (shorter than or equal to 32 bytes) and long frames (longer than 32 bytes). Assuming normal operation only one interrupt occurs for one frame and transfer direction: RME-interrupt for receive direction and XPR-interrupt for transmit direction. So the ESCC2-device driver module can be made very compact since the data transfer itself is performed by the DMA-controller ADMA.

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

4.2 ESCC2-Device Driver Module Entries

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.

4.2.1 Message Entry Point

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 the function 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 four different types of tasks are supported by the ESCC2-message entry point. These tasks are distinguished by a task or message identification, called as ID.

In the application at hand the following message IDs are supported:

ID	Action/Function
MODULE_INIT (= 0)	Escc2Init()
INIT_DATA_LINK (= 1)	InitDataLink()
ASSIGN_ADDRESS (= 2)	AssignAddress()
SEND_FRAME (= 3)	SendFrame()

The functions of the message entry point Escc2MsgEntry (CIM MSG_DESCR_PTR cmd) are shown in **figure 12**.

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

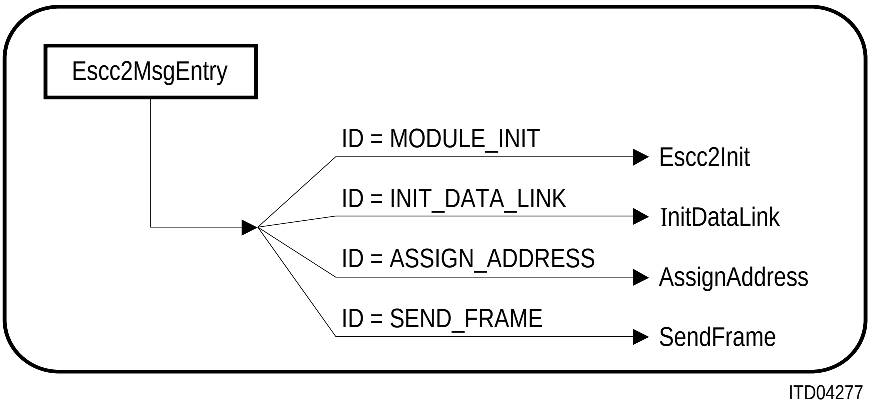


Figure 12
Message Entry Point

In the following the actions are described in detail:

Escc2Init (CIM_MSG_DESCR_PTR command) initializes the ESCC2-device as well as the local data structures. The base device address is set by executing DdshGetDeviceAddr (DEVICE1).

An important task is the initialization of the interrupt. As the basic address is known (see chapter 4.4), the registers can be accessed. At first all interrupts should be masked out by writing FFH into the Interrupt Mask Register 0,1 (IMR0, IMR1).

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

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

InitDataLink (CIM_MSG_DESCR_PTR command) contains the general initialization of all ESCC2-mode specific functions and data for one of the two data links. The data link (channel A or B) is specified by the element Entity of the structure that is accessible by pointer *command*.

Parameter 1 of the CIM_MSG_DESCR structure accessible by the pointer *command*, selects the receive mode (HDLC-auto mode/transparent mode). The clock mode is represented by parameter 2. By writing 60H 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-, XDU- and XPR-interrupts in HDLC-mode corresponding to the DMA-transfer mode. Then the HDLC-receiver and transmitter are reset.

Moreover the corresponding DMA-channels of the ADMA are initialized. For receive direction a message buffer (Dlc → dmaRcMessage) is allocated and the appropriate ADMA-channel program is started.

AssignAddress (CIM_MSG_DESCR_PTR command) 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 are used for 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. After the execution of this function only frames with the programmed address are received, assuming that the ESCC2 doesn't operate in transparent mode.

SendFrame (CIM_MSG_DESCR_PTR command) inserts the *command* into the transmit queue of the current data link structure and executes the transmitter action corresponding to the current state of the transmitter.

4.2.2 Interrupt Entry Point

The interrupt entry point is introduced during initialization of the ESCC2-module in function Escc2Init (…).

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). The interrupt requests the CPU to read status information from the ESCC2 and to execute appropriate interrupt service routines. 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).

Since the SAB 80C166 doesn't support the interrupt acknowledge cycle interrupt polling mode is used. As a result there is only one interrupt entry.

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 (**see figure 13**).

After entering the interrupt function Escc2Interrupt(), the General Interrupt Status Register as well as the channel specific Interrupt Status Register are analyzed. Then the action corresponding to the specific kind of interrupt is executed (e.g. TxAction[…][…](), ServeRme(), …).

INTRET releases the interrupt logic of the SAB 80C166 to be able to accept further interrupts.

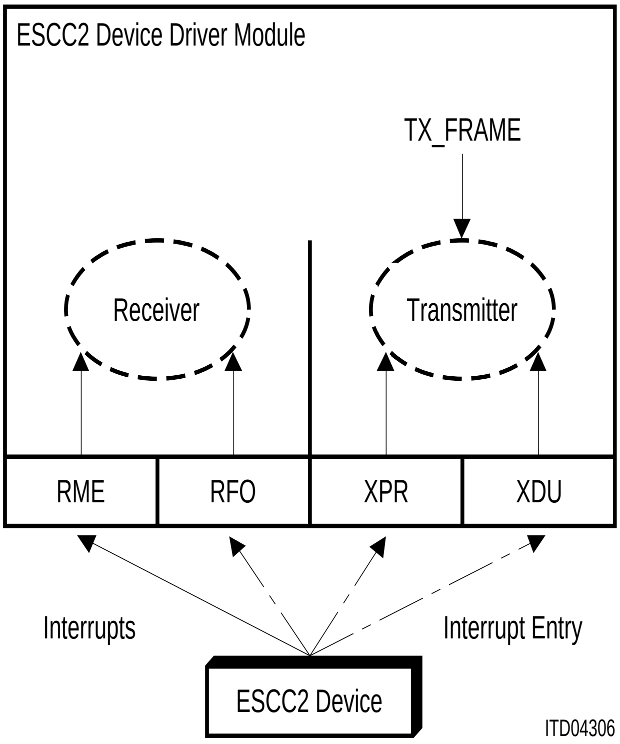


Figure 13
Interrupt Entry Point

A more detailed description of the receiver and the transmitter will follow in section 4.4.

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.

4.3 Receiver and Transmitter

In DMA-mode the receiver and transmitter of the ESCC2-device driver module is easier to design than in interrupt mode since it has not to be distinguished between short and long frames and the data transfer itself is performed by the DMA-controller.

For the receiver no event driven finite state machine is necessary. The receiver routines are reduced to ServeRme() and ServeRfo().

Since the transmitter organizes the send requests in a linked list a finite state machine is introduced to guarantee a fast access to the next frame if there is one.

4.3.1 Receiver

In DMA-mode there are only two reasons to activate the receiver: RME-interrupt and RFO-interrupt.

If a RME-interrupt occurs, the function ServeRme() is executed. At first the length of the received frame is read from the RBC-register. The pointer to the received data is buffered in *Dlc → message* and a new buffer *Dlc → dmaRcMessage* has to be allocated for subsequent frames. Since the frame length is unknown a priori a data buffer with the maximum length is allocated. With this buffer the DMA-controller is started again; finally the RMC-command is set, to enable the generation of further receiver DMA-requests.

After that the receive status byte at the end of the frame is analyzed; 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.

The RFO-interrupt is served by ServeRfo(). In this case an error message is generated and the DMA-controller is started again with the same buffer.

Dlc → dmaRcMessage allocated before. Finally the HDLC-receiver of the ESCC2 is reset.

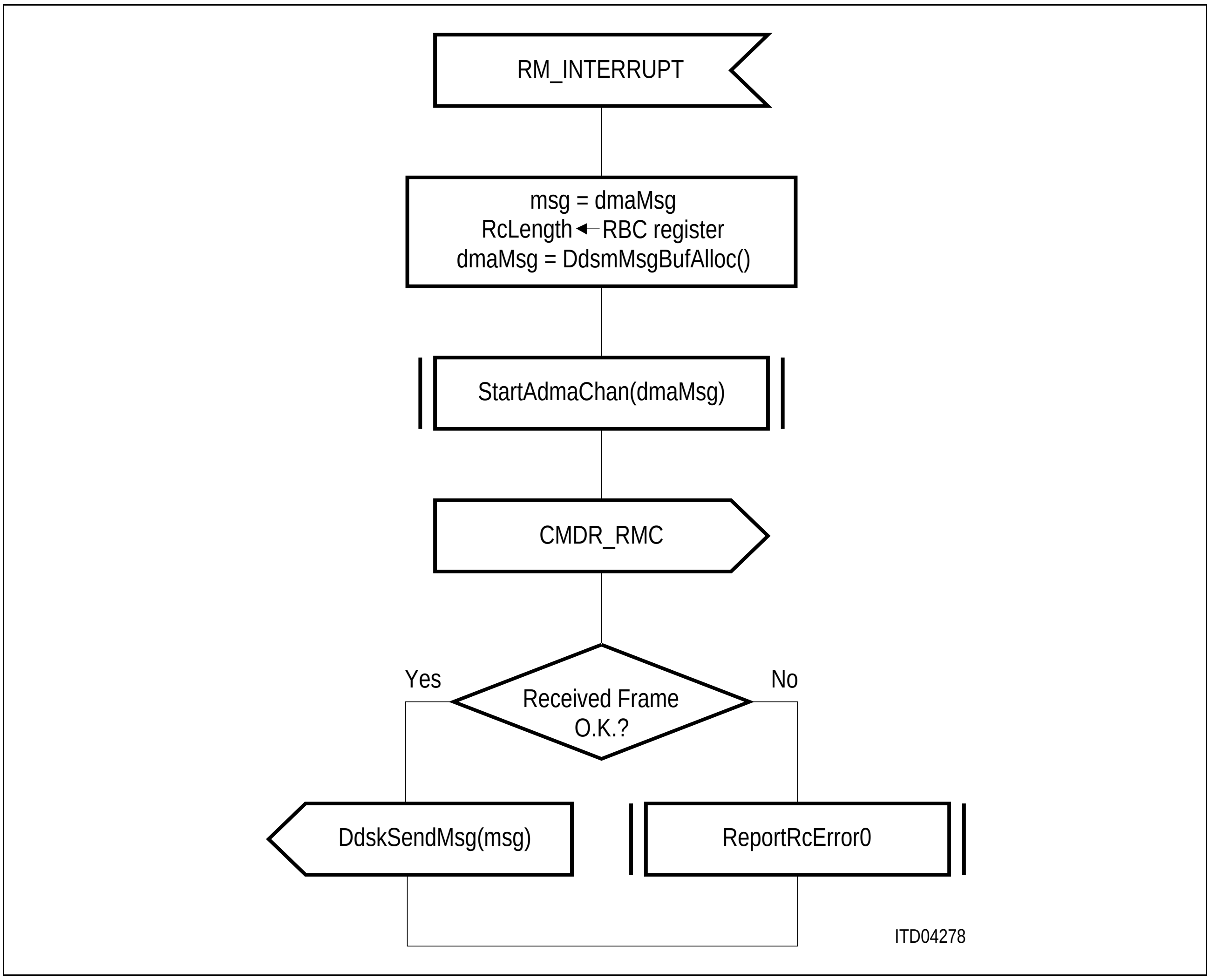


Figure 14
ServeRme()

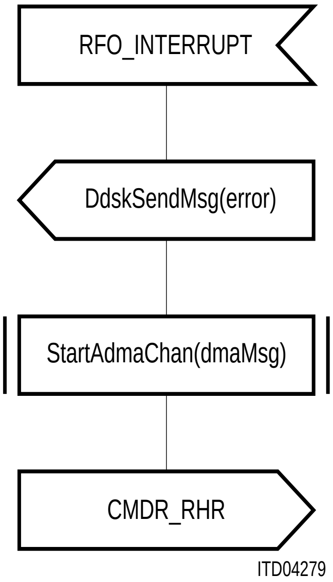


Figure 15
ServeRfo()

4.3.2 Transmitter

The transmitter is realized as a finite state machine. Three events affect the transmitter: XPR-interrupt, XDU-interrupt and the event TX_FRAME.

At the end of the initialization of the data link (see InitDataLink (…)) the current state of the transmitter is set to IDLE. After that the HDLC-transmitter of the ESCC2 is reset. In response to the reset an XPR-interrupt is generated. That means that the transmitter is ready.

The event TX_FRAME is derived from the application request SEND_FRAME (see SendFrame (…)). Since it may happen that the transmitter gets a new task while it is still busy with the previous one, at first the new task is queued in a linked list by executing the macro APPEND_TO_LIST (…). Then the appropriate action corresponding to the current state of the transmitter is executed.

Event State Table

The event state combinations are grouped in an event state table; only for clarity's sake this table is divided into two parts. The first part shows the event caused by the application: TX_FRAME. Different states corresponds to different lines, the event is represented by the column. Each element of the table includes the appropriate action and the subsequence transmitter state of a certain event-state combination.

	TX_FRAME
SENDING	NoAction (…) SENDING
READY	TxFrame (…) SENDING
IDLE	NoAction (…) IDLE

	XPR_INTERRUPT	XDU_INTERRUPT
SENDING	ServeNextFrame (…) READY (*1)	ServeXdu (…) IDLE
READY	NoAction (…) READY	NoAction (…) IDLE
IDLE	SetReady (…) READY	NoAction (…) IDLE

(*1) READY is only be taken on if the transmit queue is empty. Otherwise the next frame will be transmitted and the state will be unchanged.

The actions are grouped in a field of function pointers, called TxAction […] […]. At first during the initialization TxAction […] […] is set to *NoAction* for all elements of the array. Then, corresponding to the table the appropriate action is entered, for example:

```
…
TxAction [IDLE] [XPR_INTERRUPT] = SetReady;
```

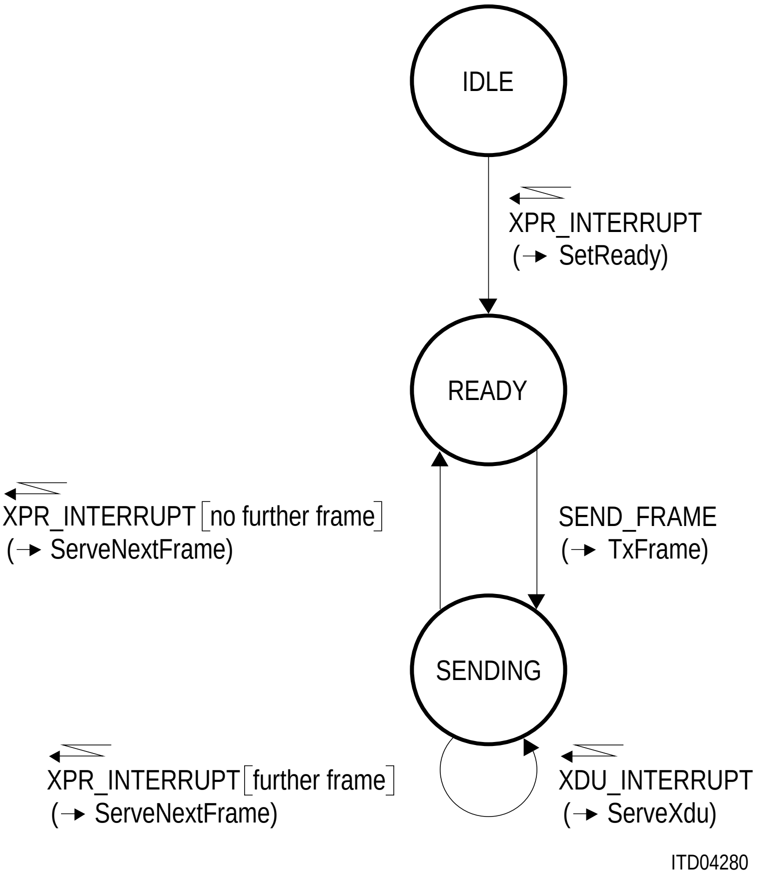


Figure 16
Transmitter – Finite State Machine

Finite State Machine

Since the transmitter is realized as a finite state machine – corresponding to the event state table discussed before – the transmitter is always in a specific state at every instant of time. From each state there are transitions to other states. Transitions occur when some event takes place. Every event causes a certain action. **Figure 16** shows the finite state machine.

SDL-Diagrams

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

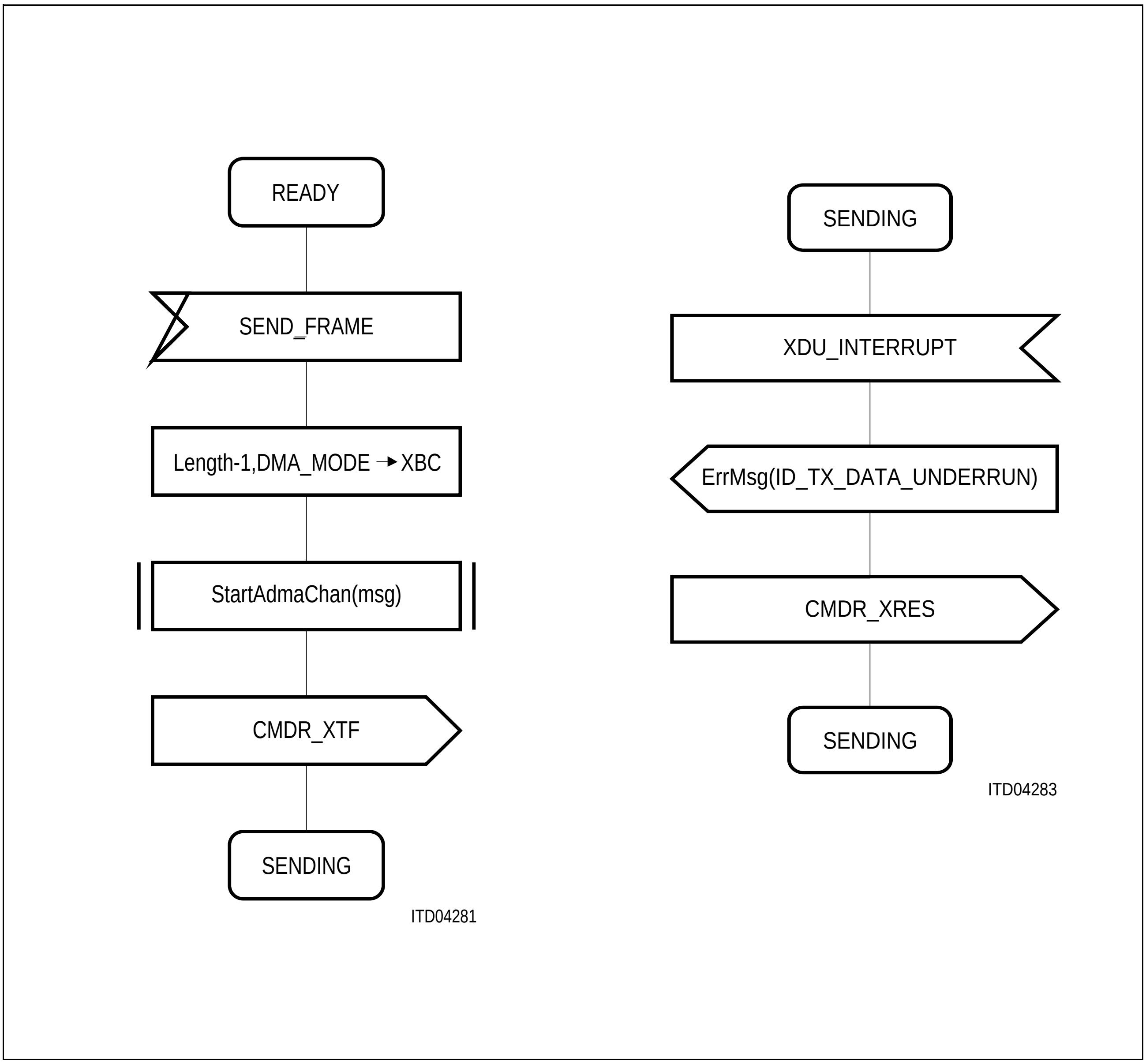


Figure 17
Transmitter Service Functions: *TxFrame ( ), ServeXdu ( )*

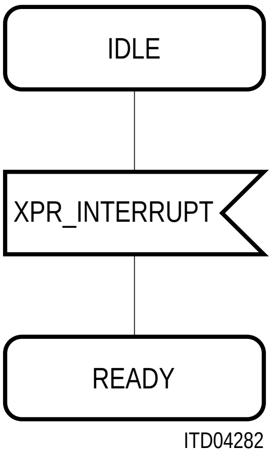


Figure 18
Transmitter Service Function: *SetReady ( )*

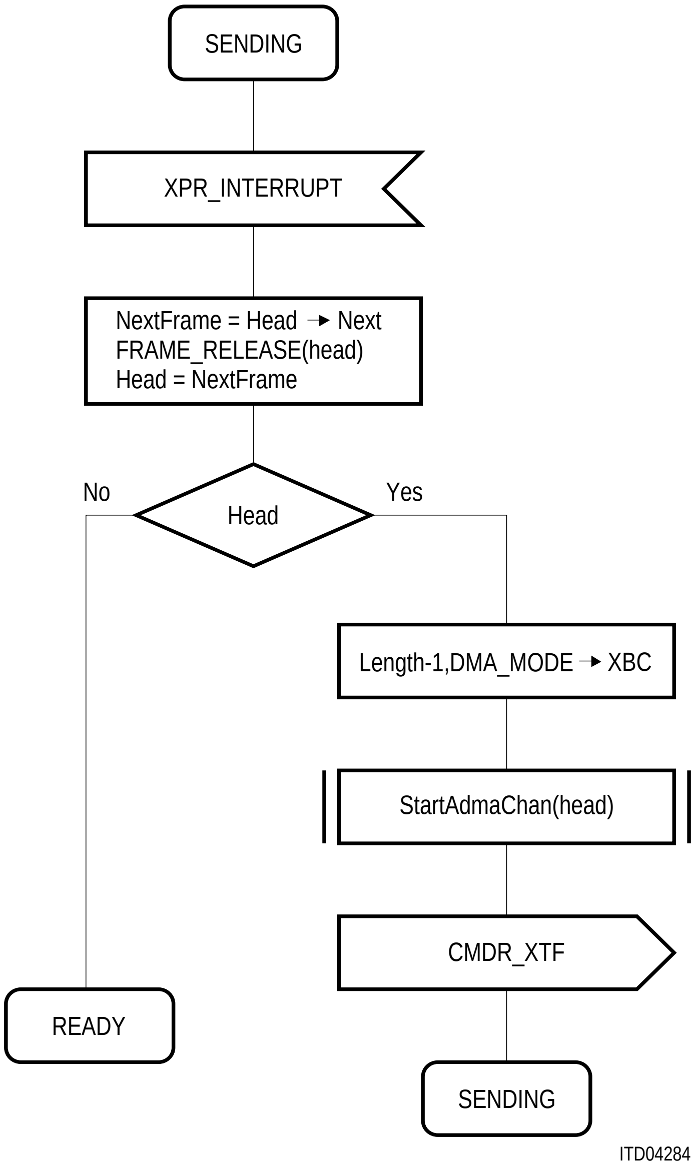


Figure 19
Transmitter Service Function: *ServeNextFrame ( )*

4.4 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 grouped 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* Escc2: The first problem when programming the ESCC2 is the access to its registers. For the sake of clarity the registers are grouped 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
{
        WORD16      fifo [16]; /* RFIFO             */
        WORD8       star; /* Status Register        */
        WORD8       rsta; /* Receive Status Reg.    */
        WORD8       mode; /* Mode Register          */
        WORD8       timr; /* Timer Register         */
        WORD8       xad1; /*Transmit Address        */
.
.
.
        WORD8       gis; /* Global Int. Status      */
        WORD8       ipc; /* Interrupt Port Conf.    */
        WORD8       isr0; /* Interrupt Status 0     */
        WORD8       isr1; /* Interrupt Status 1     */
        WORD8       pvr; /* Port Value Register     */
        WORD8       pis; /* Port Interrupt Status   */
        WORD8       pcr; /* Port Conf. Reg.         */
        WORD8       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;
```

In the application on hand only the register set for HDLC-mode is used. Now that the relative locations (= 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 initialized during the Escc2Init (…) routine. One gets the base address of the device in memory by executing DdshGetDeviceAddress (device). Assuming that the ESCC2 corresponds to DEVICE1 the following statements are executed in Escc2Init (…):

```
.
.
ESCC2_REG_MAP*                          base;
.
.
base                                    = DdshGetDeviceAddress (DEVICE1);
DataLinkCtrl [ESCC2_CH_A].ESCC2 = base;
base                                    = (ESCC2_REG_Map far*)
                                          ( ( (WORD32) 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).

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

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, HSHFR).

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

CIM_MSG_DESCR_PTR dmaRcMessage: Before an ESCC2-channel receives a frame, a message buffer has to be allocated for the DMA-controller; *dmaRcMessage* is the pointer to the channel’s current receive message buffer.

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

CIM_MSG_DESCR_PTR message: Assuming that an ESCC2-channel has received the end of a frame, the pointer *dmaRcMessage* of the message buffer allocated before is buffered in message for subsequent handling. At the same time a further buffer (*dmaRcMessage*) is allocated for the DMA-controller to receive subsequent frames.

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 and taken away from the head. 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. *Tail* is controlled by sending frames to the ESCC2-channel; *head* is controlled by the XPR-interrupt service routine.

CIM_MSG_DESCR_PTR tail: See CIM_MSG_DESCR_PTR *head*.

int TxState: The serial interface of the ESCC2 consists of two full duplex channels. So both channels are able to transmit data at the same time. Normally a transmit procedure consists of a transmit request (generated by the application), a DMA-transfer and finally an XPR-interrupt generated by the ESCC2-device; in this case various states are be taken on. The current states are stored in *TxState.*

4.5 ADMA-Device Driver Module

As described in section 4.1 the ADMA-device driver module provides nine service routines to control the ADMA-device and the data transfer:

- InitAdmaDevice (…)
- InitAdmaChan0 (…)
- InitAdmaChan1 (…)
- InitAdmaChan2 (…)
- InitAdmaChan3 (…)
- StartAdmaChan0 (…)
- StartAdmaChan1 (…)
- StartAdmaChan2 (…)
- StartAdmaChan3 (…)

Since the DMA-channels 0 and 1 or DMA-channels 2 and 3 provide the same services (ADMA 0/1 <–> receiver of ESCC2-channel A/B; ADMA 2/3 <–> transmitter of ESCC2-channel A/B; the description can be reduced to five functions which are described in more detail in the following:

InitAdmaDevice (CIM_MSG_DESCR_PTR command) initializes the ADMA-device as well as the local data. The base device address is set by executing DdshGetDeviceAddr (DEVICE2). The addresses of the channel specific registers are evaluated using the base device address and appropriate offsets. Similar to the ESCC2-module the registers are grouped in data structures: ADMA_GEN_REGS (ADMA-General Registers) and ADMA_CHAN_REGS (ADMA-Channel Registers).

By programming the General Mode Register of the ADMA the operating mode is selected. In the application on hand local mode is selected; furthermore the physical bus width is set to 16 bit, all interrupts are masked and rotating priorities are programmed. No limit is programmed for continuous bus cycle; the General Delay Register is set to 0. So no limit is set to the extent the ADMA can load the bus.

InitAdmaChan0 (CIM_MSG_DESCR_PTR command) initializes the channel program for ADMA-channel 0 consisting of two command blocks.

There are two basic types of channel commands:

– type 1 channel commands for data transfers and
– type 2 channel commands for command chaining control.

A complete channel program consists of at least two channel command blocks, one with a type 1 command and one with a type 2 command. So the data structure chan0Program (ADMA channel 0 Program) contains the two elements cB1ST1 (command block 1 of short type 1) and cB2T2 (command block 2 of type 2) **(see figure 20)**.

cB1ST1 specifies the actual data transfer operation and contains parameters such as source pointer, destination pointer, byte count, etc… cB2T2 specifies the control operations of the channel, as for example channel stop operations.

ADMA-channel 0 serves the DMA-requests of the receiver of ESCC2-channel A. Initializing the corresponding channel program the following settings are defined:

– the data transfer is source synchronized,
– data is transferred in words,
– the destination pointer has to be incremented,
– the source/destination pointer resides in the memory space,
– RFIFO of channel A is source,
– there is a maximum length of frame buffer.

cB2T2 completes the channel program with a conditional stop, if byte count is reached.

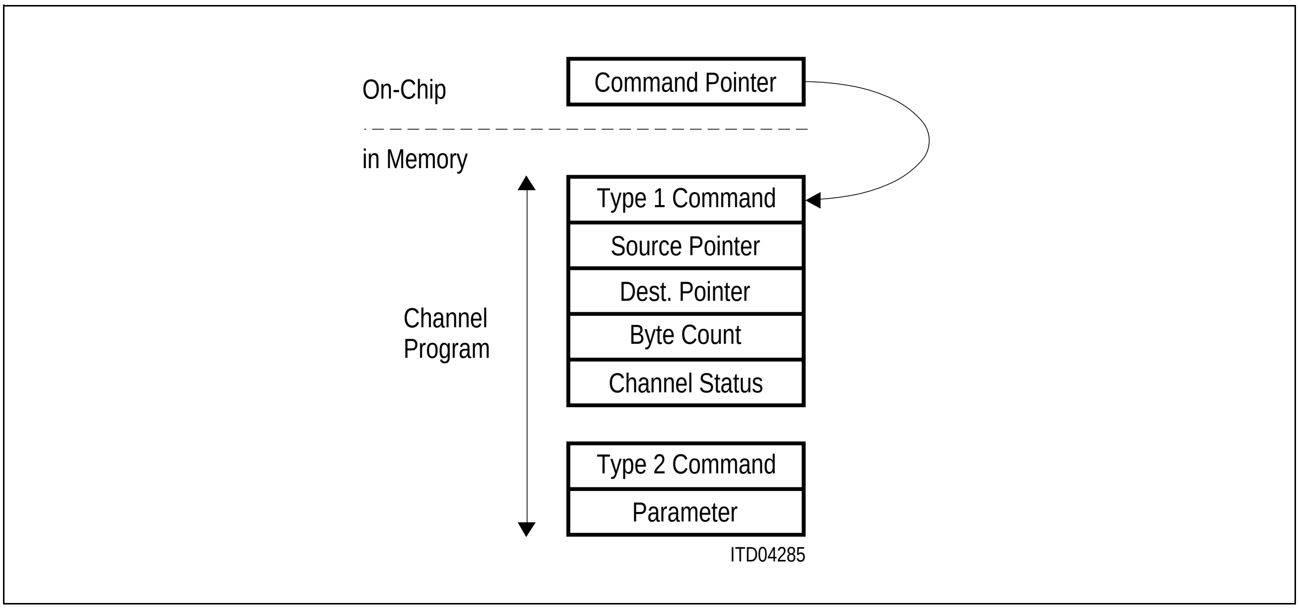


Figure 20
Simplest Channel Program

At the end of the routine InitAdmaChan0 (…) the address of channel program is buffered.

InitAdmaChan1 (CIM_MSG_DESCR_PTR command) is very similar to InitAdmaChan0 (…), with the following differences:

– the data transfer is destination synchronized,

– the source pointer has to be incremented,

– XFIFO of channel A is destination.

StartAdmaChan0 (CIM_MSG_DESCR_PTR command) enters the destination pointer to the corresponding parameter of *cB1ST1* in *chan0Program* and sets the address of the channel program to the corresponding Command Pointer Register. Then the channel program is started by programming the General Command Register appropriately.

StartAdmaChan1 (CIM_MSG_DESCR_PTR command) enters the source pointer to the corresponding parameter of *cB1ST1* in *chan1Program* and sets the address of the channel program to the corresponding Command Pointer Register. Then the channel program is started by programming the General Command Register appropriately.

The service routines InitAdmaChan0/1/2/3 (…) are executed once during the initialization of the data link; the short routines StartAdmaChan0/1/2/3 (…), however, are executed every time a frame has been received or has to be transmitted.

4.6 Software Workaround for Single-Cycle Mode

As mentioned in section 2.2 a problem arises when ADMA and ESCC2 operate in single cycle mode. Since the trailing edge of the DRRn (Receiver DMA-Request) is received late, a continuous request is assumed and the ADMA executes an unwanted transfer.

This failure occurs every time when the RFIFO is read by the DMA-controller; so for frames shorter or equal to 32 bytes a further byte/word is transferred at the end of the frame. For frames longer than 32 bytes additionally every 32 bytes one spurious byte/word is transferred **(see figure 21)**.

Since the trailing edge of DRRn cannot be used to terminate the DMA-transfer in time the software workaround starts from the idea to control the DMA-transfer by byte counter. The DMA-controller provides a feature called data chaining, which allows to transfer data to predefined locations until the byte counter reaches 0 **(see figure 22)**.

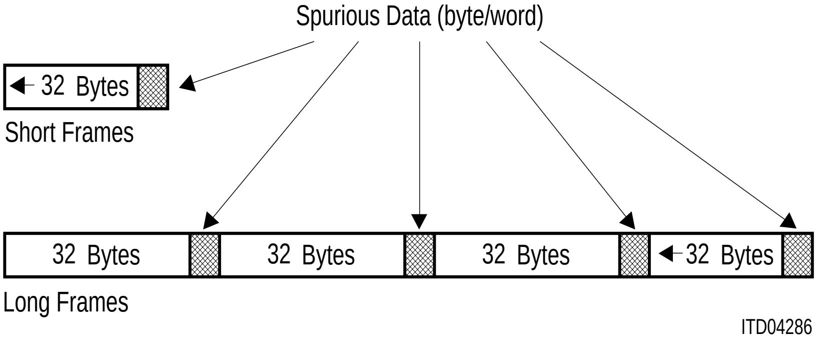


Figure 21
Spurious Transfers

In the application at hand the predefined locations can be organized continguously in memory and the byte counter should correspond to the FIFO-length of 32 bytes. After transferring 32 bytes the ADMA stops data transfer due to byte count end and reads the following data chain list element from memory containing the next data pointer and byte counter. During this time the ESCC2 resets the DRRn-signal and no spurious data transfer will occur. Using this method the additional transfer every 32 bytes of one byte/word can be avoided. At the end of a frame with n bytes (n mod 32 is not equal to 0), however, one additional transfer is unavoidable. Regarding this failure the data buffer must be greater than the maximum receive length; finally the real length of the frame received can be evaluated from the RBC-register during the ServeRme() routine.

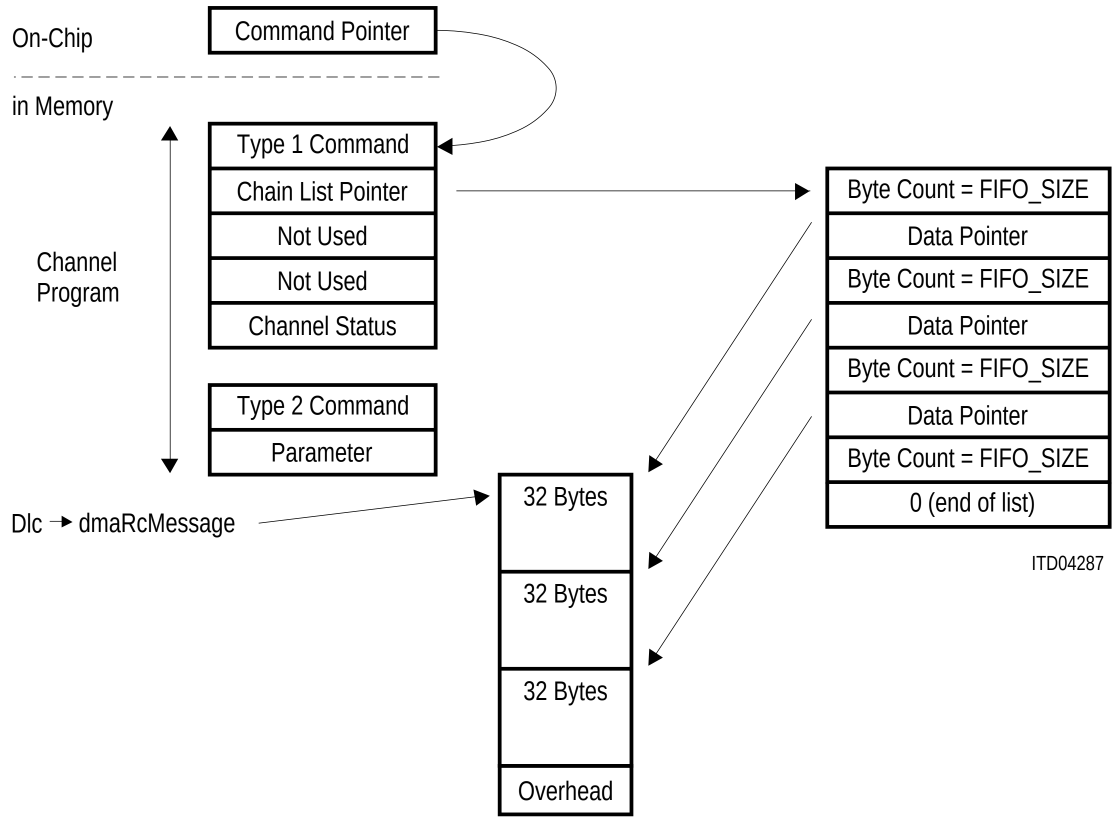


Figure 22
Data Chaining

5 The Application Module HSHFR

The application module HSHFR contains an high speed HDLC-frame relay. The function of this application is very simple: Frames received on ESCC2-channel A are transmitted via ESCC2-channel B and vice versa.

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_PTRmsg_entry).

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

5.2 Detailed Description of the Application Module

In addition to the function DdsIntegrate() and the message entry point HshfrMsgEntry (CIM_MSG_DESCR_PTR command) the application module contains two routines, which initialize the module or perform the relay switch of the frames: HshfrInit (…) and HshfrRcFrameOk (…).

HshfrInit (CIM_MSG_DESCR_PTR command) sends messages to the ESCC2-device driver module to initialize the data links (ID = INIT_DATA_LINK) and to assign the appropriate address (ID = ASSIGN_ADDRESS). The channel identification (Entity), receive mode (P1) and clock mode (P2) are passed as elements of the message. After the initialization the application module receives messages from the ESCC2-device driver whenever a frame has been received.

One message ID is supported:
RC_HDLC_FRAME_ESCC2_OK.

If a HDLC-frame has been received the function HshfrRcFrameOk (CIM_MSG_DESCR_PTR cmd) is executed.

HshfrRcFrameOk (…): All HDLC-frames received on one channel (e.g. ESCC2-channel A) are sent transparently in HDLC-mode on the other channel (e.g. ESCC2-channel B) or vice versa.

6 Performance of the System

Discussing the performance of the system a very important question is how much time t_{applic} remains to run an application compared to the time t_{data} which is necessary to organize and execute a data transfer. These times are summarized in the total bus time t_{bus}:

$$t_{\text{bus}} = t_{\text{data}} + t_{\text{applic}} \qquad (1)$$

6.1 Time for Data Handling

Assuming that the application is performed only by the microcontroller SAB 80C166 and the data handling is executed by the microcontroller as well as by the DMA-controller ADMA one gets the following equations for t_{data} and t_{applic}:

$$\begin{aligned} t_{\text{data}}(n) &= t^{\text{ADMA}}_{\text{setup}} + t^{\text{ADMA}}_{\text{transfer}}(n) \\ &\quad + t^{\text{‘166}}_{\text{DDM}}(d) + t^{\text{‘166}}_{\text{DDS}} \qquad (2) \end{aligned}$$

$$t_{\text{applic}} = t^{\text{‘166}}_{\text{applic}} \qquad (3)$$

n = frame length in number of bytes;

d = direction rx/tx;

$t^{\text{ADMA}}_{\text{setup}}$: Setup time of ADMA after being started by the microcontroller.

$t^{\text{ADMA}}_{\text{transfer}}$ (n): Time to transfer data including bus arbitration; this time depends on the frame length.

$t^{\text{‘166}}_{\text{DDM}}$ (d): Time to organize the data transfer (list linking, buffer allocation, interrupt handling, programming the devices, …) in the device driver module; this time depends on the direction (transmit/receive).

$t^{\text{‘166}}_{\text{DDS}}$: Time to route a received frame from the device driver module to the application module or the frame which has to be transmitted from the application module to the device driver module.

The following times have been measured:

		$f^{\text{ADMA}}_{\text{ext}}$	$f^{\text{'166}}_{\text{ext}}$
		25 MHz	**40 MHz**
$t^{\text{ADMA}}_{\text{setup}}$	$33 \times \frac{2}{f^{\text{ADMA}}_{\text{ext}}}$	2.64 µs	
$t^{\text{ADMA}}_{\text{arb}}$	$10.5 \times \frac{2}{f^{\text{ADMA}}_{\text{ext}}}$	840 ns	
$t^{\text{ADMA}}_{\text{cyc}}$	$4 \times \frac{2}{f^{\text{ADMA}}_{\text{ext}}}$	320 ns	
$t^{\text{'166}}_{\text{DDM}}$ (tx)	$6 \times 10^3 \times \frac{2}{f^{\text{'166}}_{\text{ext}}}$		300 µs
$t^{\text{'166}}_{\text{DDM}}$ (rx)	$5.36 \times 10^3 \times \frac{2}{f^{\text{'166}}_{\text{ext}}}$		268 µs
$t^{\text{'166}}_{\text{DDS}}$	$7 \times 10^2 \times \frac{2}{f^{\text{'166}}_{\text{ext}}}$		35 µs

$t^{\text{ADMA}}_{\text{arb}}$: Arbitration time from DMA-request to first transfer cycle (without code prefetch: setup time).

$t^{\text{ADMA}}_{\text{cyc}}$: Time to transfer one byte/word (8/16-bit access) in two cycle mode.

Especially the time used by the SAB 80C166 depends on the microcontroller itself, the specific application and the way how the software is realized. For the sake of simplifying the discussion

$t^{\text{'166}}_{\text{DDM}}$ is taken as the maximum of the time used for receive or transmit direction:

$$t^{\text{'166}}_{\text{DDM}} = MAX\,(t^{\text{'166}}_{\text{DDM}}\ (\text{rx}),\ t^{\text{'166}}_{\text{DDM}}\ (\text{tx})\ )$$

$$= t^{\text{'166}}_{\text{DDM}}\ (\text{tx})$$

The transfer time $t^{\text{ADMA}}_{\text{transfer}}$ (n) is calculated as:

$$t^{\text{ADMA}}_{\text{transfer}}\ (\text{n}) = \frac{\text{n}}{32} \times \text{T}_{\text{transfer}} + t_{\text{rest}} \qquad (5)$$

where $T_{transfer} = t_{arb}^{ADMA} + (32)16 \times t_{cyc}^{ADMA}$ is the time to transfer a 32-byte pool of data using (8-bit) 16-bit access; n/32 is the quotient (number of 32-byte pools to be transferred) and n mod 32 is the remainder.

If n mod 32 = 0 t_{rest} = 0, if not up to 31 bytes have to be transferred at the end of a frame. The time for the transfer of up to 31 bytes t_{rest} is calculated by summarizing the time for bus arbitration t_{arb} after DMA-request and the number of data transfer cycle times t_{cyc} using (8-bit) 16-bit access:

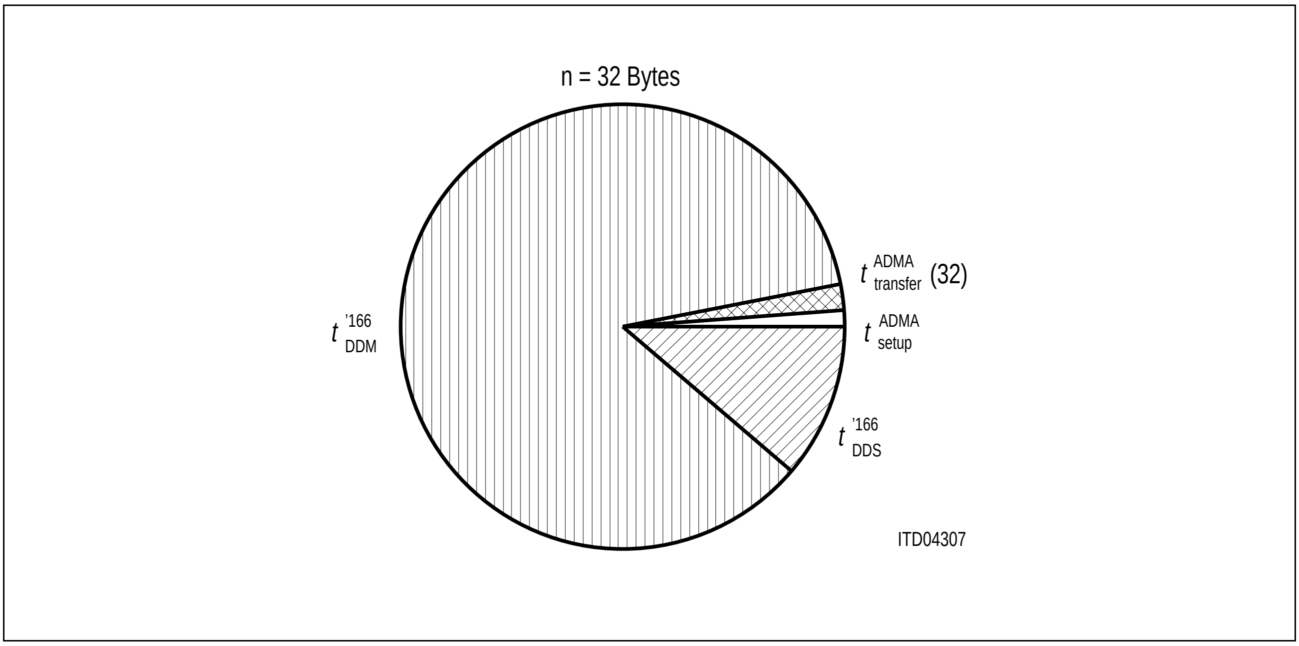


Figure 23
Transfer Time as a Part of the Time Used for Data Handling n = 32 Bytes

$$t_{rest} = t_{arb} + \frac{\text{n mod 32}}{(1)\ 2} \times t_{cyc} \qquad (4)$$

The **figures 23-25** represent how the time t_{data} (100 %) is divided up depending on the frame length. They show that the time to transfer frames of length equal to or shorter than 32 bytes is negligible compared to the time for organizing the data handling in the device driver module. Independent of the frame length, the setup time of the ADMA t_{setup}^{ADMA} is negligible. One gets the following equation for t_{data} (n):

$$t_{data}\,(n) \approx \frac{n}{32}\ T_{transfer} + t_{DDM}^{‘166} + t_{DDS}^{‘166}$$

$$T_{org} = t_{DDM}^{‘166} + t_{DDS}^{‘166}$$

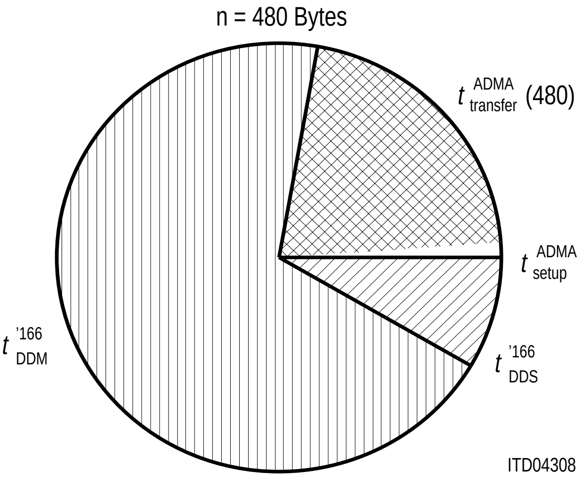


Figure 24
Transfer Time as a Part of the Time Used for Data Handling n = 480 Bytes

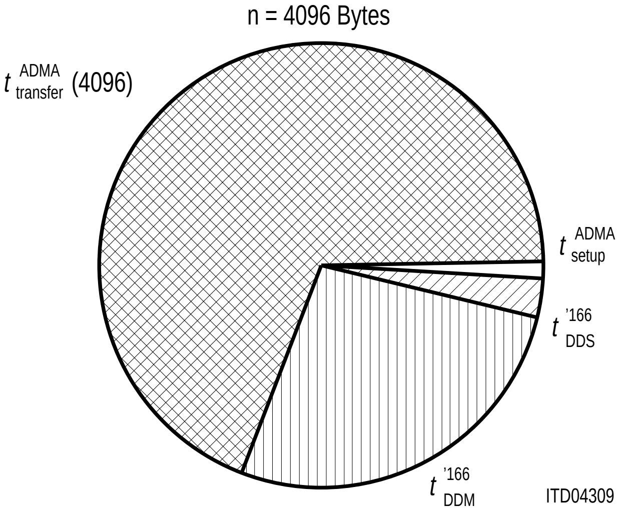


Figure 25
Transfer Time as a Part of the Time Used for Data Handling n = 4096 Bytes

T_{org} is the time to organize the data transfer in the device driver module and to route the frames from the DDM to the APM and vice versa. As mentioned above this time depends on the microcontroller, the specific application and the way how the software is realized.

6.2 Bit Rate and Frame Rate

The correspondence between bit rate and frame rate on the one hand and bus load on the other hand is present in **figures 26/27**. **Figure 26** shows the maximum bit rate the ADMA is able to cope, when DMA-transfers are requested back-to-back without delay:

$$r^{\mathrm{ADMA}}_{\mathrm{bit,max}} = \frac{32\ \mathrm{bytes}}{T_{\mathrm{transfer}}}$$

Using 25-MHz clock and 16-bit access the rate is 42.7 Mbit/s. Dividing up this performance on four DMA-channels the ADMA is able to transfer data for two bidirectional ESCC2-channel, each operating at a bit rate of 10 Mbit/s. This transfer rate has been tested successfully in the system implemented.

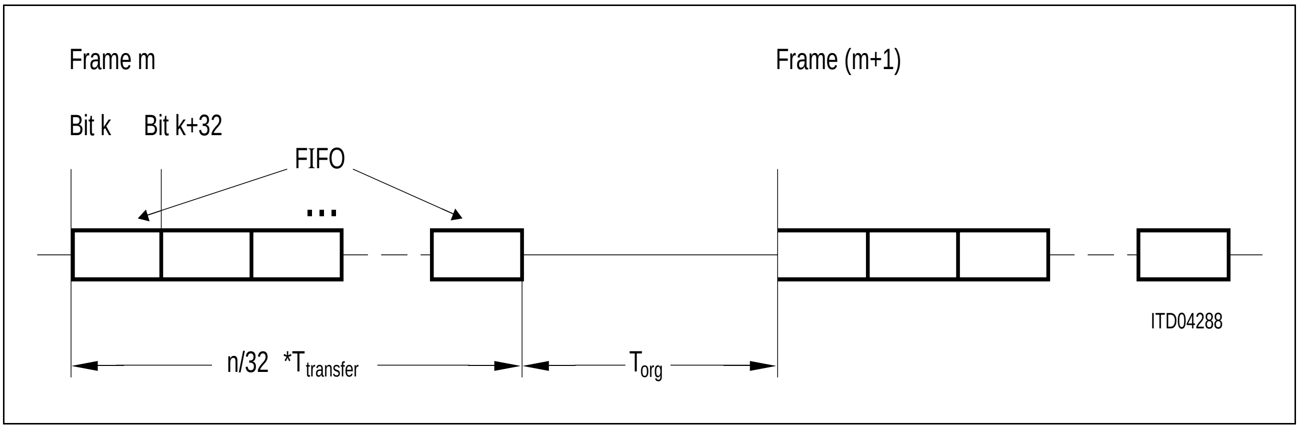


Figure 26
Maximum Bit Rate

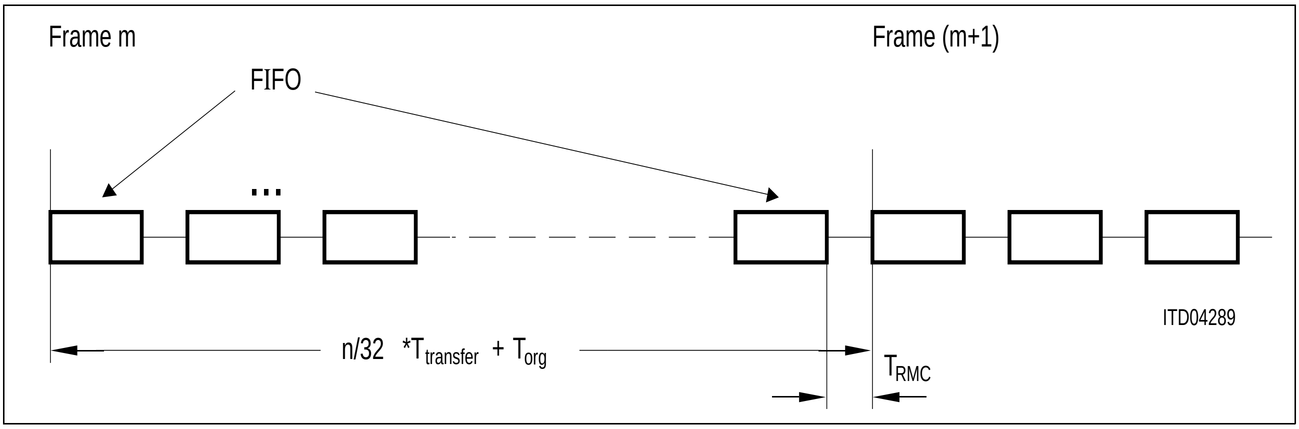


Figure 27
Maximum Frame Rate

The frame rate can be calculated as:

$$r_{\mathrm{frame}} = \frac{1}{\frac{n}{32} \times T_{\mathrm{transfer}} + T_{\mathrm{Org}}}$$

Regarding **figure 27** the following condition has to be considered when calculating the maximum frame rate. For bit rate less than the maximum bit rate the CPU is able to perform organizational processing between the data transfers. To avoid loosing data or even a complete frame in this case because of a receive data overflow or a receive frame overflow the next frame should not arrive till the microcontroller has set the RMC-command after an RME-interrupt event. Let us call the time from RME-interrupt to the RMC-command T_{RMC}, this time is a part of T_{org} and consists of reading the RBC-register of the ESCC2, allocating a data buffer for the next frame, preparing and starting the ADMA-channel program and setting the RMC-command to the ESCC2. If we assumed back-to-back frames with shared flags (worst case) then since the flow of data is constant the same time T_{RMC} has to exist between end of DMA-transfer of every pool of 32 byte and the beginning of the following pool.

So the frame rate is given by the following equation (n > 32):

$$r_{\text{frame, max}} \leq \frac{1}{\frac{n}{32} \times (T_{RMC} + T_{transfer})}$$

The corresponding bit rate for **continuous** frame transfer and maximum frame rate is (ADMA: 25 MHz, 16-bit access; 80C166: 40 MHz; T_{RMC} = 190 µs):

$$r_{\text{bit}} \leq \frac{32 \times 8\ \text{bit}}{T_{RMC} + T_{transfer}} = 1.3\ \frac{\text{Mbit}}{\text{s}}$$

Obviously the bottleneck for continuous frame transmission is the time T_{RMC}. Again it has to be considered that the time T_{RMC} depends on the microcontroller, the specific application and the way of software realization. To guarantee continuous frame transmission with 10 Mbit/s T_{RMC} would have to be equal or shorter than 19.6 µs.

The main part of the time T_{RMC} is used for converting addresses from 80C166 mode into ADMA-mode. This has to be done every time a pointer is written to the ADMA's-registers or channel program.

To reduce the time for address converting it is recommended to introduce predefined linked list, which contain pointers in 80C166 mode as well as in ADMA-mode. In this case T_{RMC} can be reduced by nearly 65 %. This method reduces the time for organizational processing and allows a higher frame rate.

6.3 Time for Application

Up to now the performance is analyzed without considering the time for the execution of applications.

Going back to equation (1) for t_{applic} one gets the relative part of the total bus time in the following manner:

$$t_{\text{applic}} = t_{\text{bus}} - t_{\text{data}}$$

Given a frame rate of r_{frame} the time to organize and execute the data transfer over the parallel bus is $\frac{n}{32} \times \mathrm{T}_{\text{transfer}} + \mathrm{T}_{\text{org}}$. This leaves a bus capacity for running the application equal t_{applic} per frame:

$$t_{\text{applic}} = \frac{1}{r_{\text{ frame}}} - \left(\frac{n}{32} \times \mathrm{T}_{\text{ transfer}} + \mathrm{T}_{\text{ org}}\right)$$

In the **figure 28** the time for application is depicted for several frame lengths and frame rates.

Example: t_{applic} = 100 µs (free bus capacity per frame) remains to run an application for one frame (e.g. LAN-frame) with a length of 480 bytes (n/32 = 15) and a frame rate of nearly 2000 frames/s.

6.4 Performance Improvements

In chapter 5.2 it has been shown that the ADMA (normal mode, 16-bit access, two cycle mode) is able to cope the maximum bit rate of the ESCC2 (2 bidirectional channels, 10 Mbit/s). The bottleneck is the bus load used by the microcontroller for data handling. To reduce the time used by the CPU another microprocessor can be used or the software has to be realized appropriately (linked list to avoid pointer conversions).

To improve the data transfer and therefore to reduce the bus load caused by the ADMA one can operate in single-cycle mode. This transfer mode is described in chapter 2. Corresponding to the software work around described in chapter 3 one can achieve an performance improvement of nearly 30 % regarding only the data transfer.

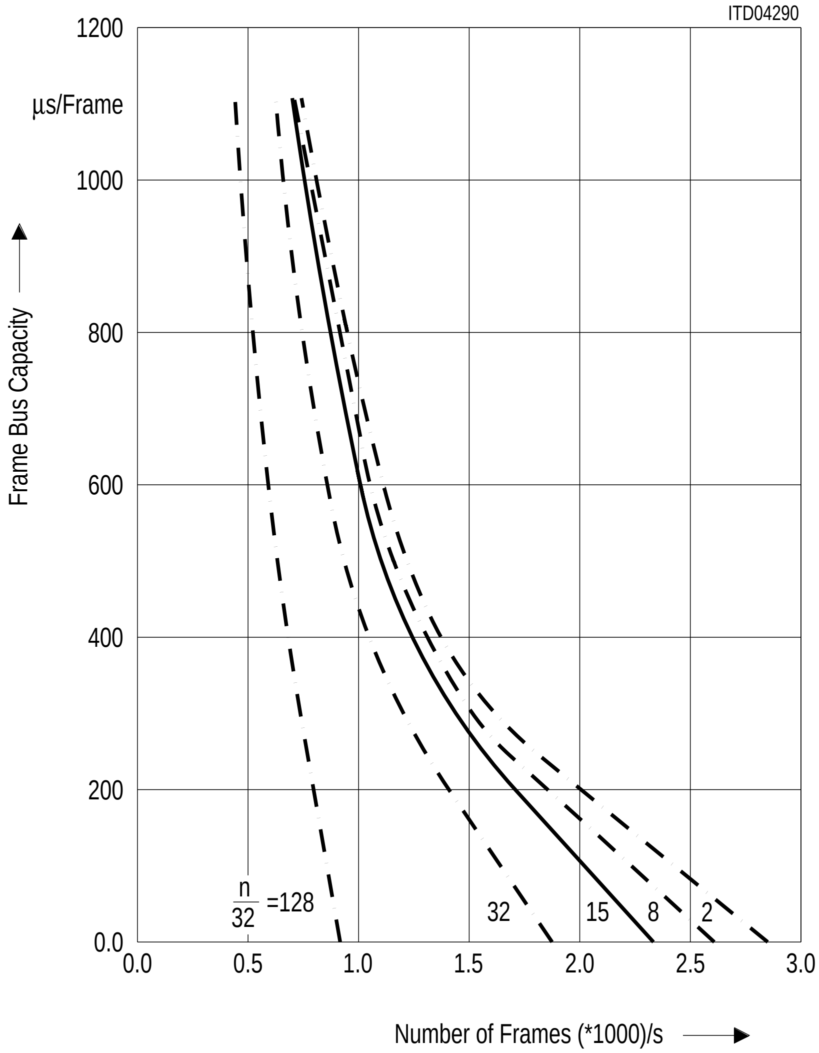


Figure 28
Free Bus Capacity per Frame

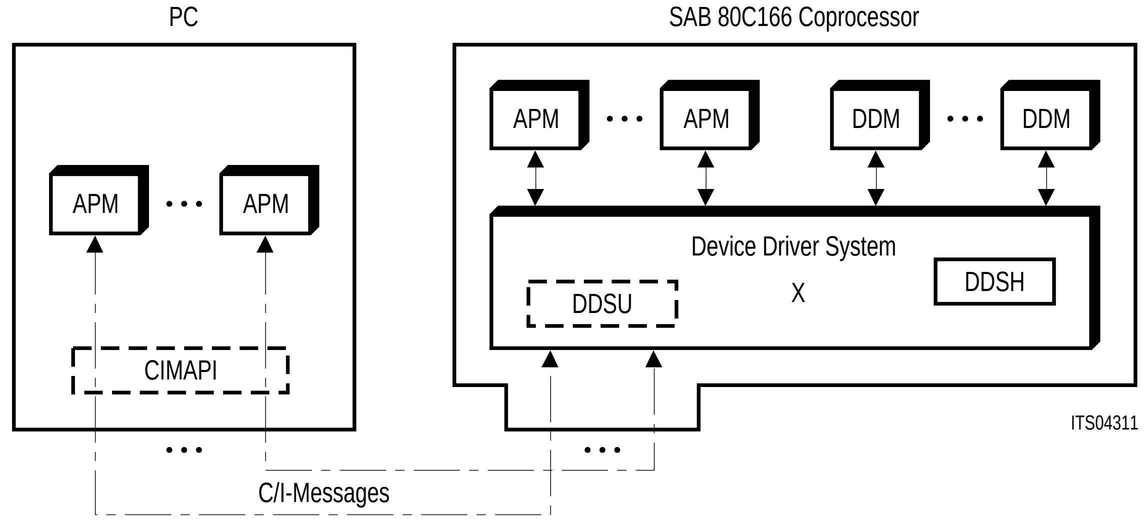
7 Device Driver System

This chapter gives an overview about the system software used to implement the application specific software modules. This Device Driver System (DDS) 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.

7.1 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 29**.



DDSU	:=	Device Driver System User Interface
DDSH	:=	Device Driver System Hardware Adaption
APM	:=	Application Program Module
DDM	:=	Device Driver Module
CIMAPI	:=	C/I-Message Application Program Interface

Figure 29
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 30**.

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

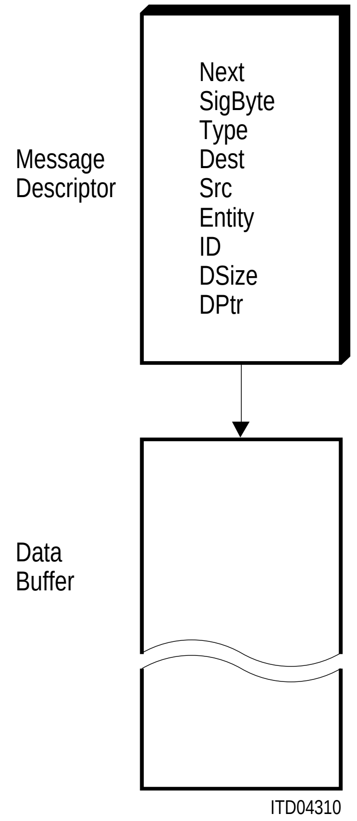


Figure 30
C/I-Message Format

7.2 General Module Architecture

A module to be integrated into the device driver system's environment must fulfill only a minimum set of requirements. The figure below 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 allows every module 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 6.5, are needed inside a module (e.g.: to request and send C/I-messages, en-/disable interrupts, …).

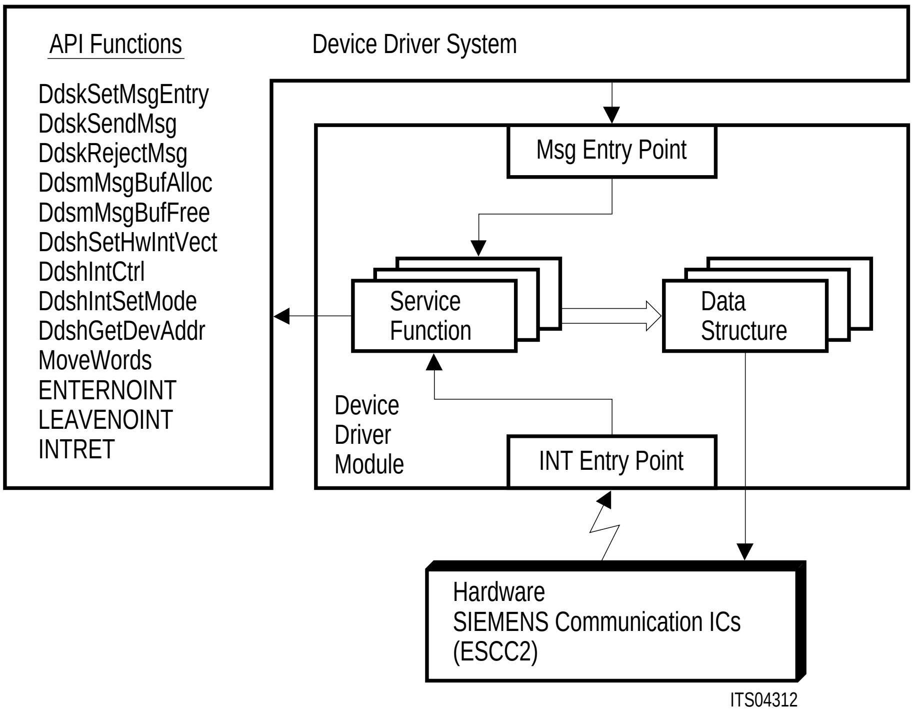


Figure 31
General Module Architecture

7.3 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.**

7.4 The Example HSHFR

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

A functional block diagram of the complete example is shown in **figure 32**.

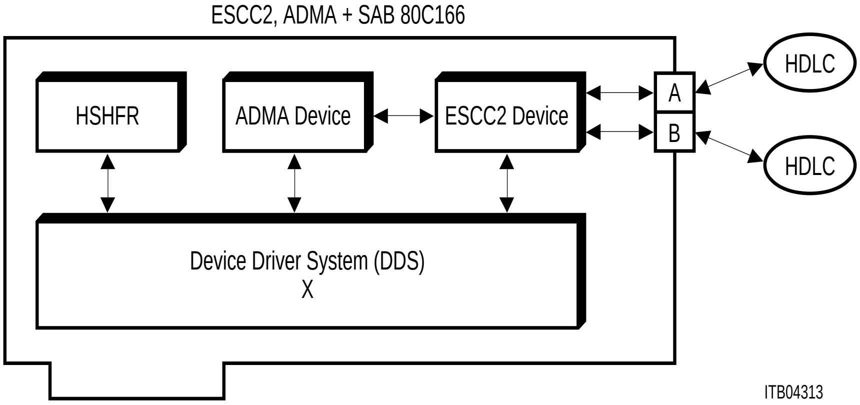


Figure 32
Structure of Relay Software

To concentrate on the device driver software and the application of the ESCC2 with ADMA in a communication system, only a minimum interface to the device driver system and the hardware specific adaptions (see section 6.5 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 33** shows the information flow inside the whole communication subsystem, especially how the different software layers (modules) are involved. The routing from channel A to channel B is illustrated, assuming an incoming HDLC-frame. The ESCC2-device driver module handles the RME-interrupt (see chapter 3), 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 HSHFR-module. The HSHFR-module reacts by routing the frame received on channel A to channel B and sending a SEND_FRAME message to the ESCC2-device driver module. One element of the C/I-message, the parameter entity (**see figure 32**), determines the ESCC2-channel to be used for transmission and the source for a received frame.

The data buffer of the routed C/I-message remains the same (DPtr; for more details refer to chapter 4).

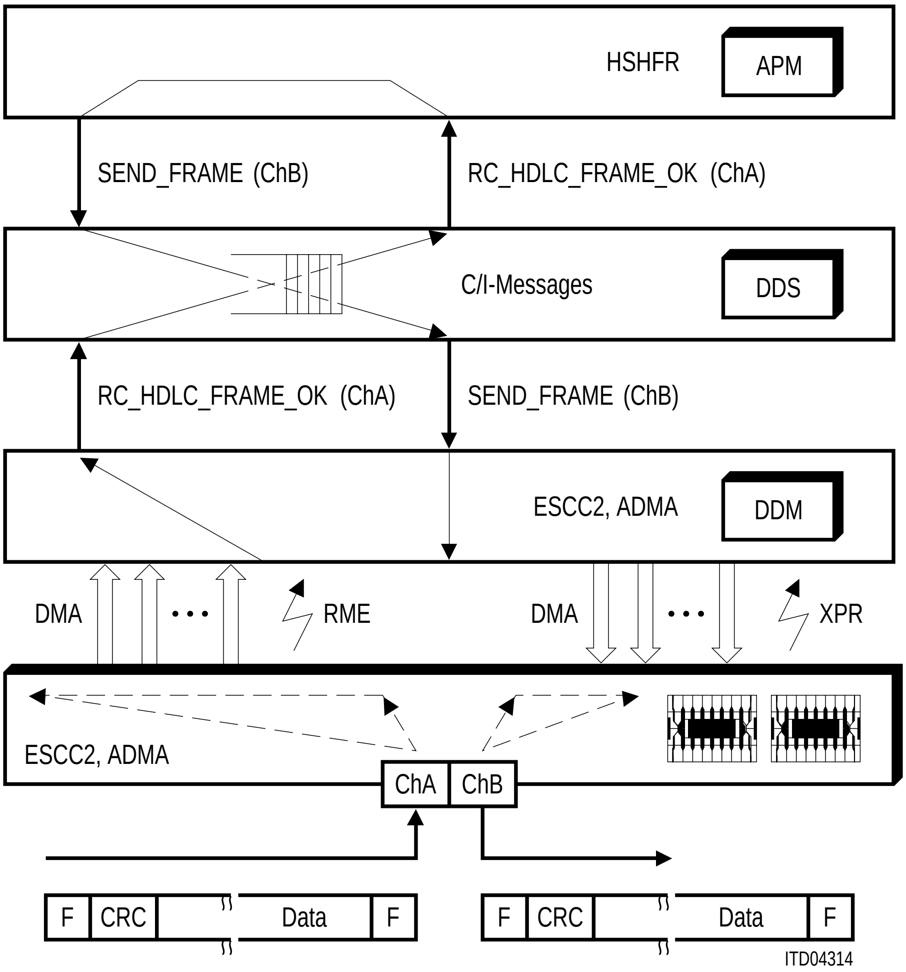


Figure 33
Information Flow from Channel A to Channel B

7.5 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 a 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 of 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 C/I-message *ciMsg* just received 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 on the processor type (e.g.: 80C166).

DdshIntCtrl (HW_INT_TYPE intx, BOOL enable);

Enables or disables an individual interrupt source.

DdshIntSetMode (HW_INT_TYPE intx, INT_MODE_TYPE mode);

Sets interrupt mode. In this application only device 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 specfied device *dev*. Only device 1 is supported in this application.

MoveWords (W16PTR srcPtr, 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.

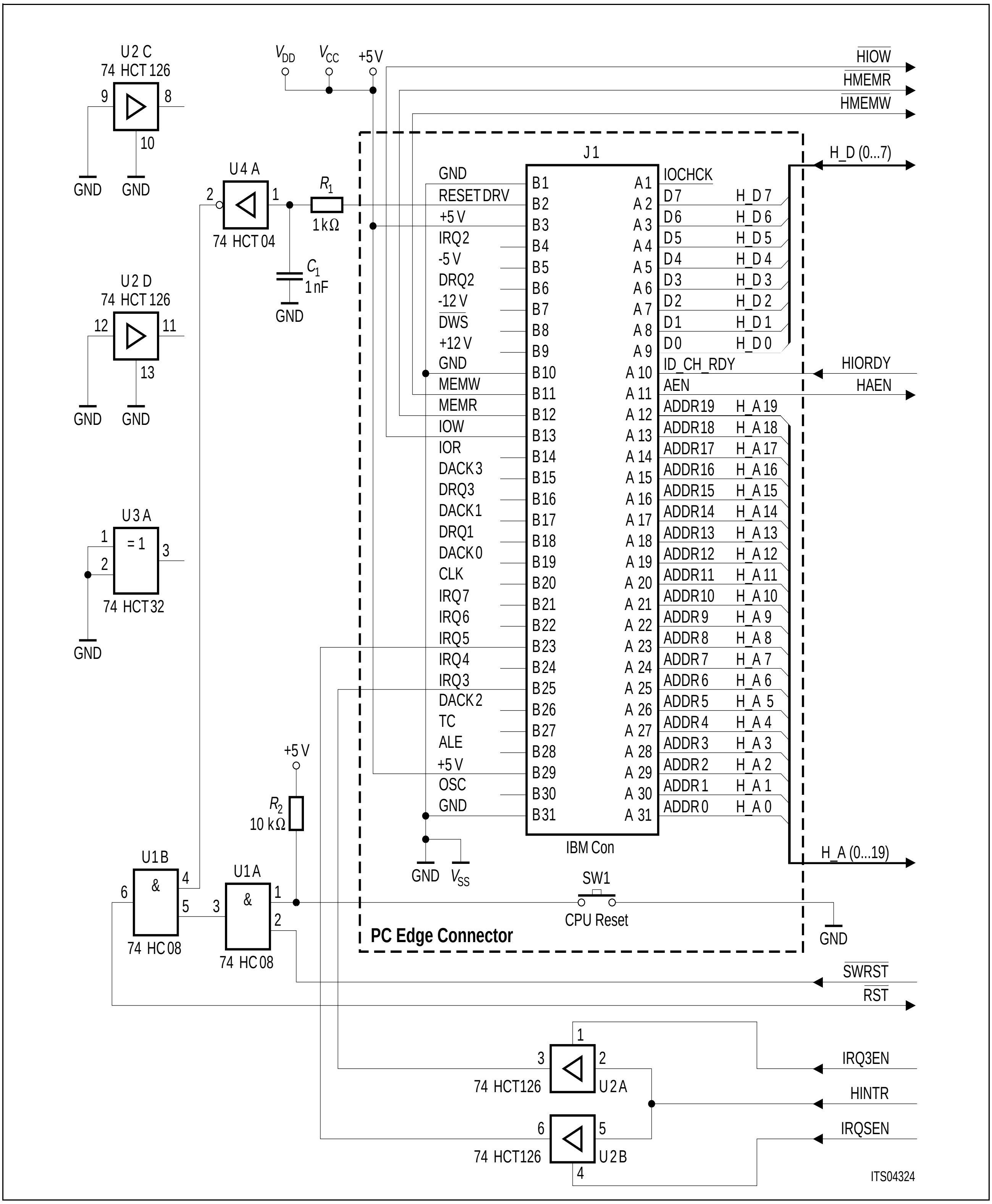
LEAVENOINT (WORD cpuState);

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.

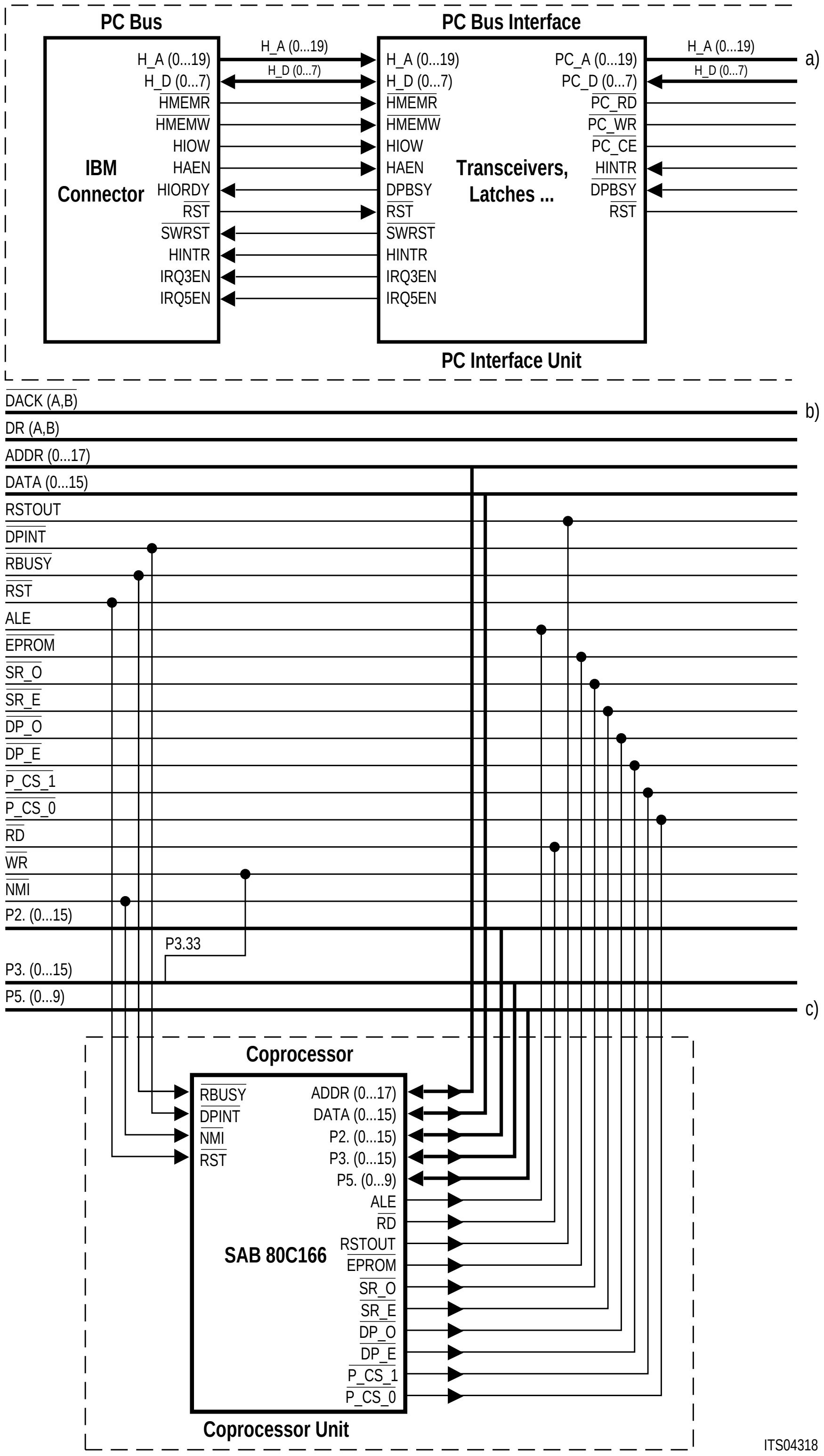
INTRET
Is a special macro defined in DDSH. It releases the interrupt logic of the corresponding interrupt line. This is necessary to allow the interrupt controller to accept interrupts. Should be called whenever an interrupt routine is left.

8 Appendix

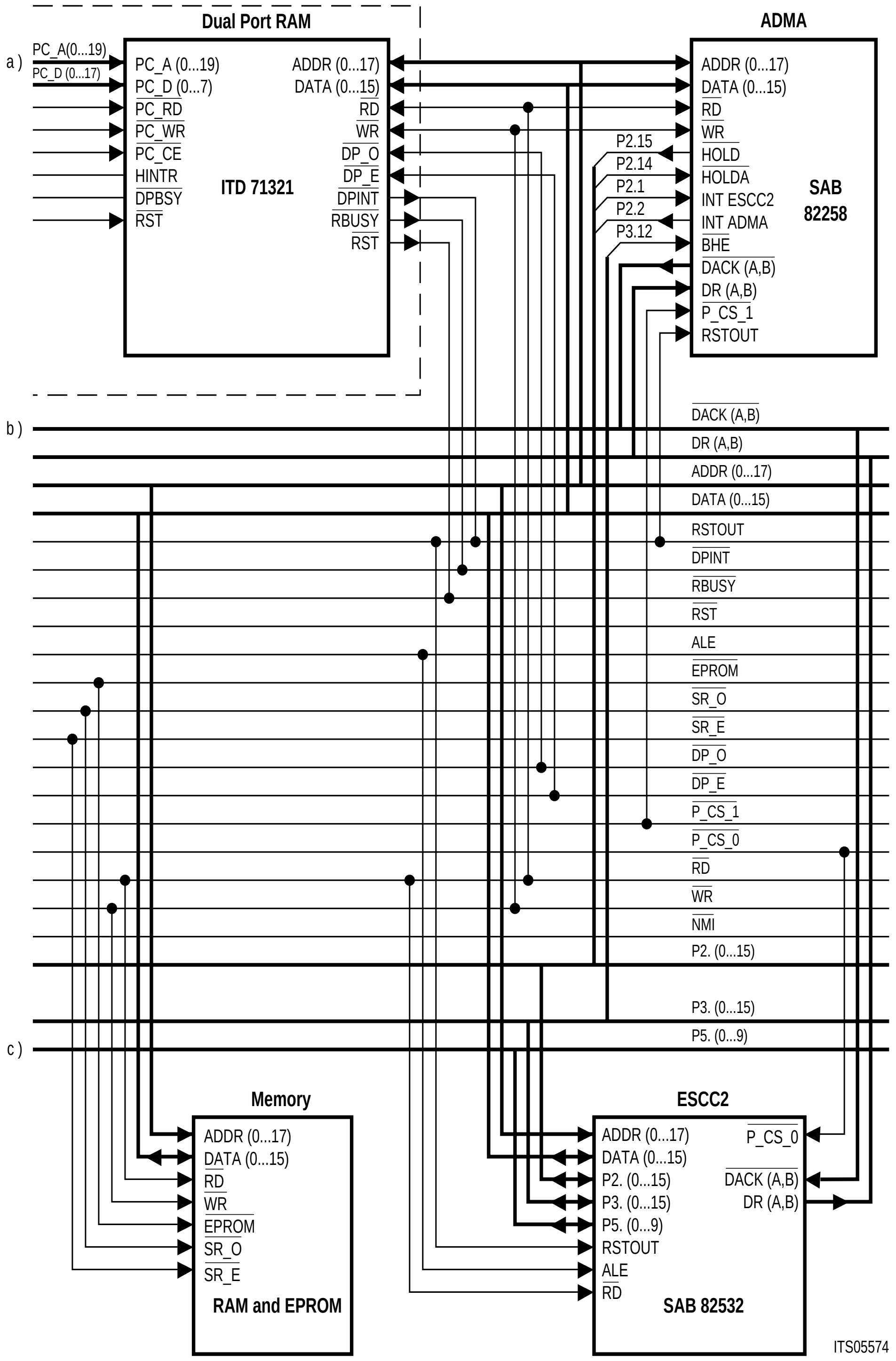
8.1 Appendix A



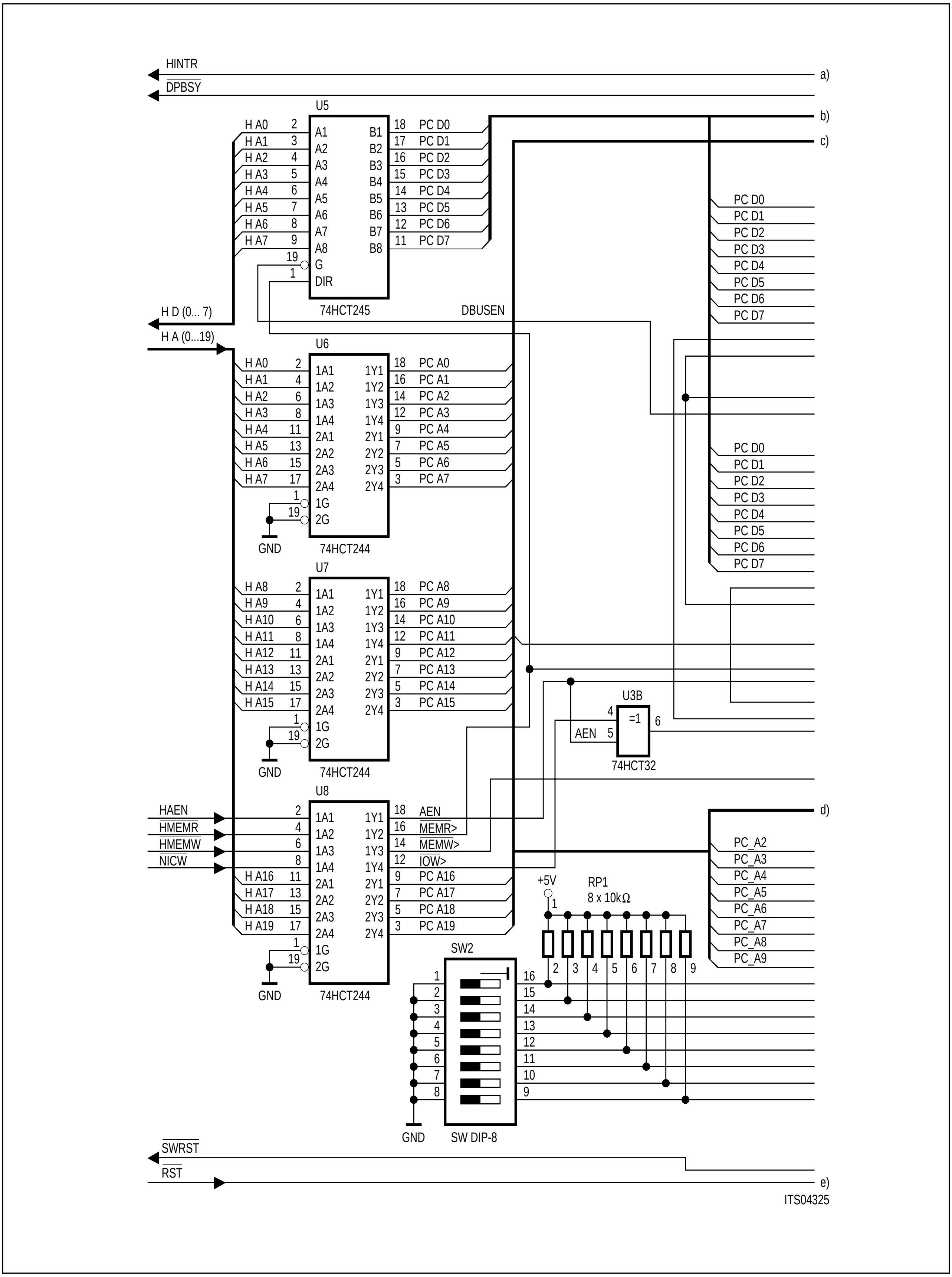
PC-Host Bus



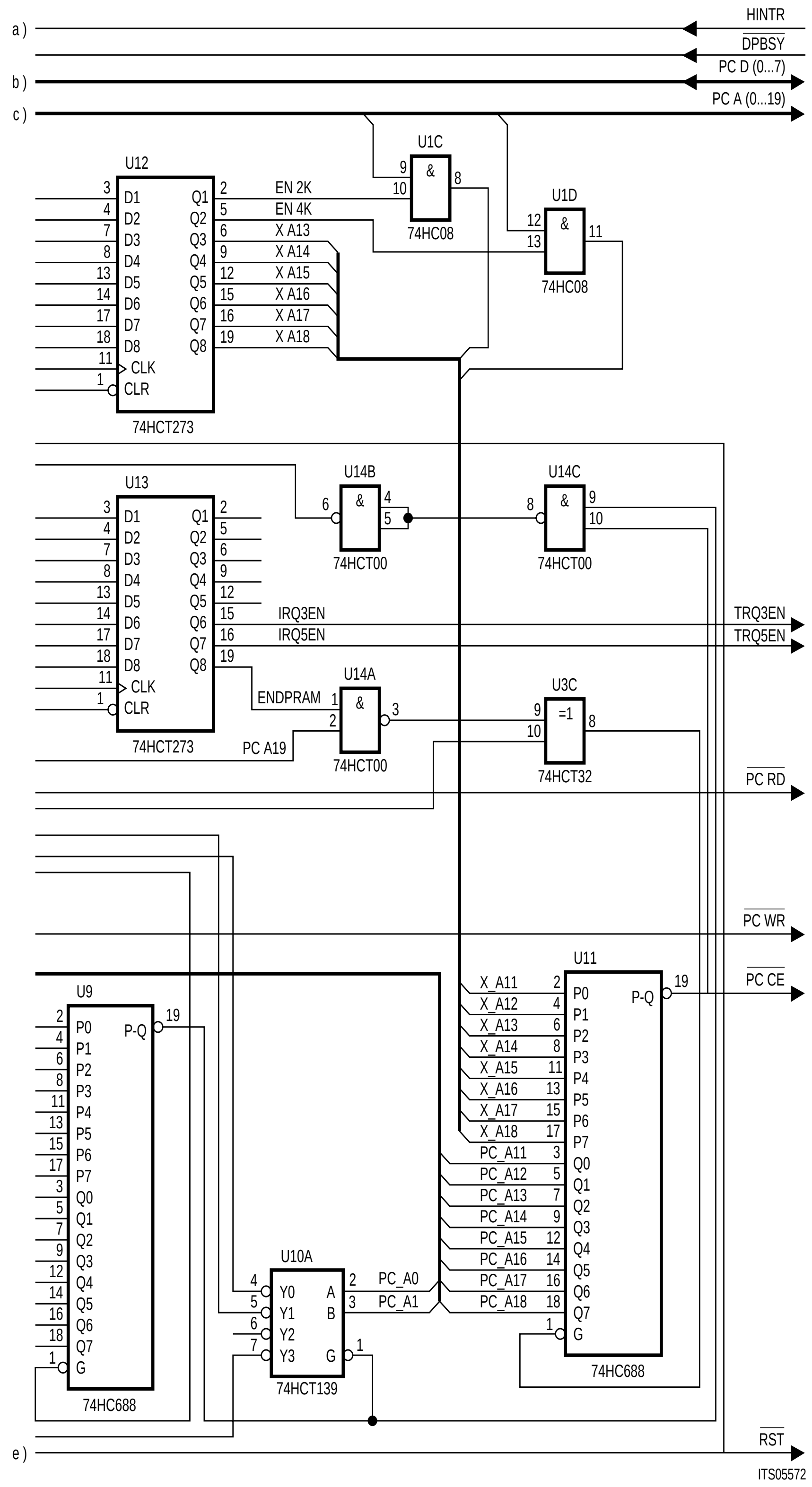
Main Block Diagram



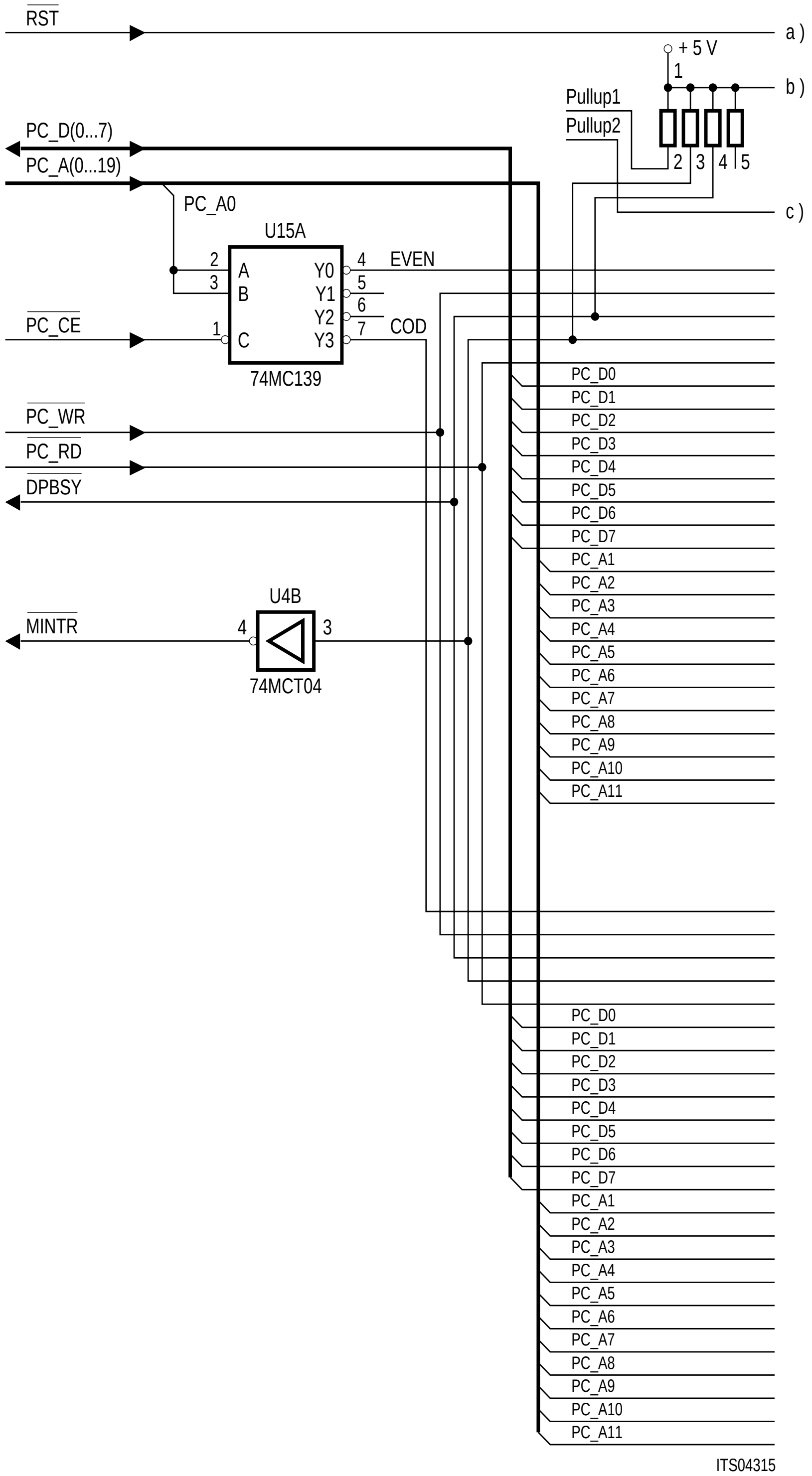
Main Block Diagram (cont'd)



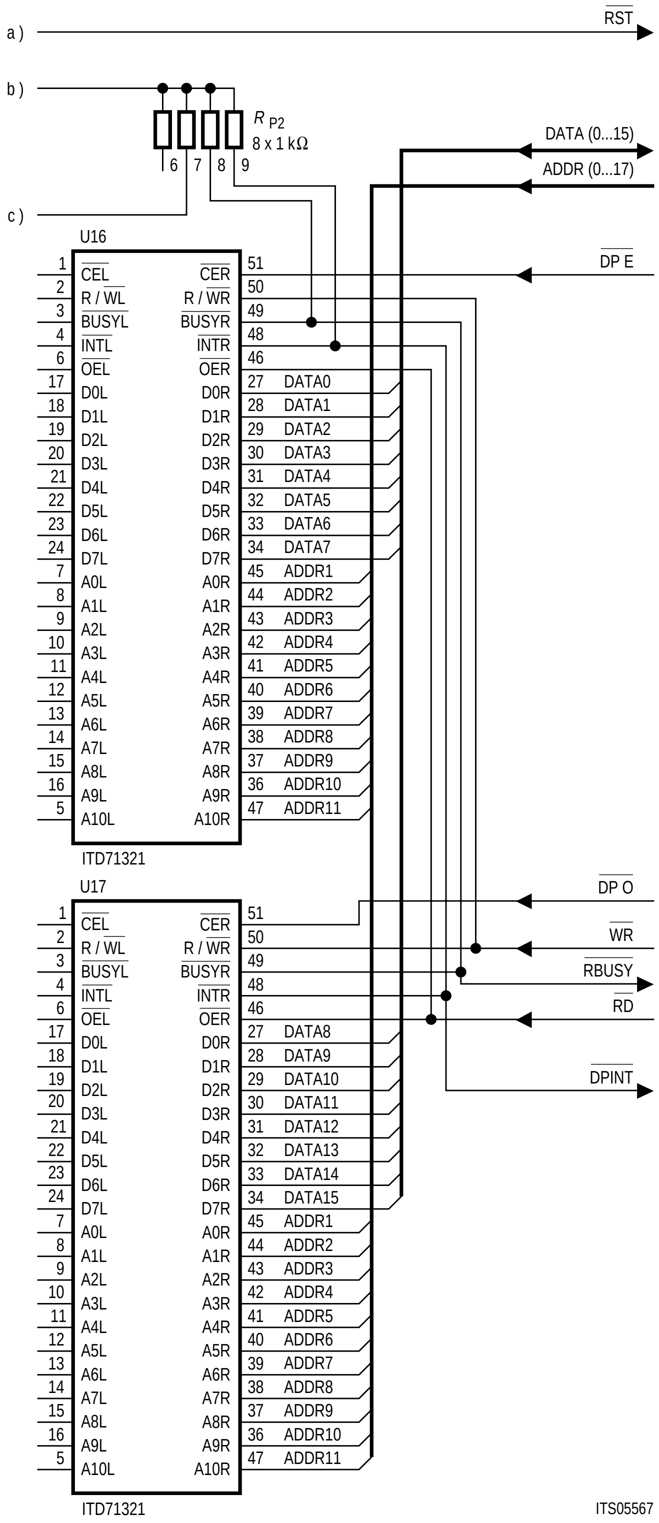
PC-Bus Interface-Logic



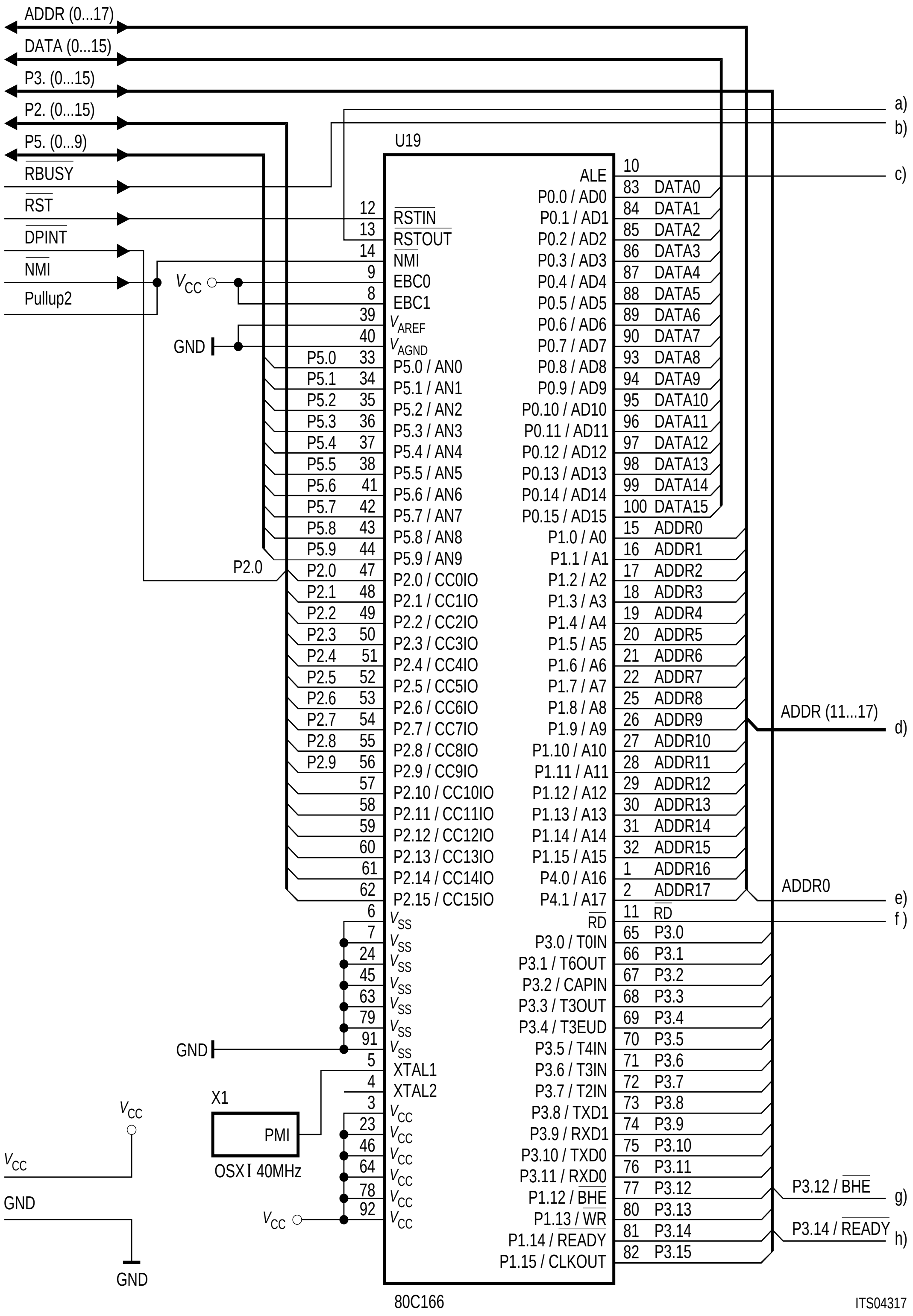
PC-Bus Interface-Logic (cont’d)



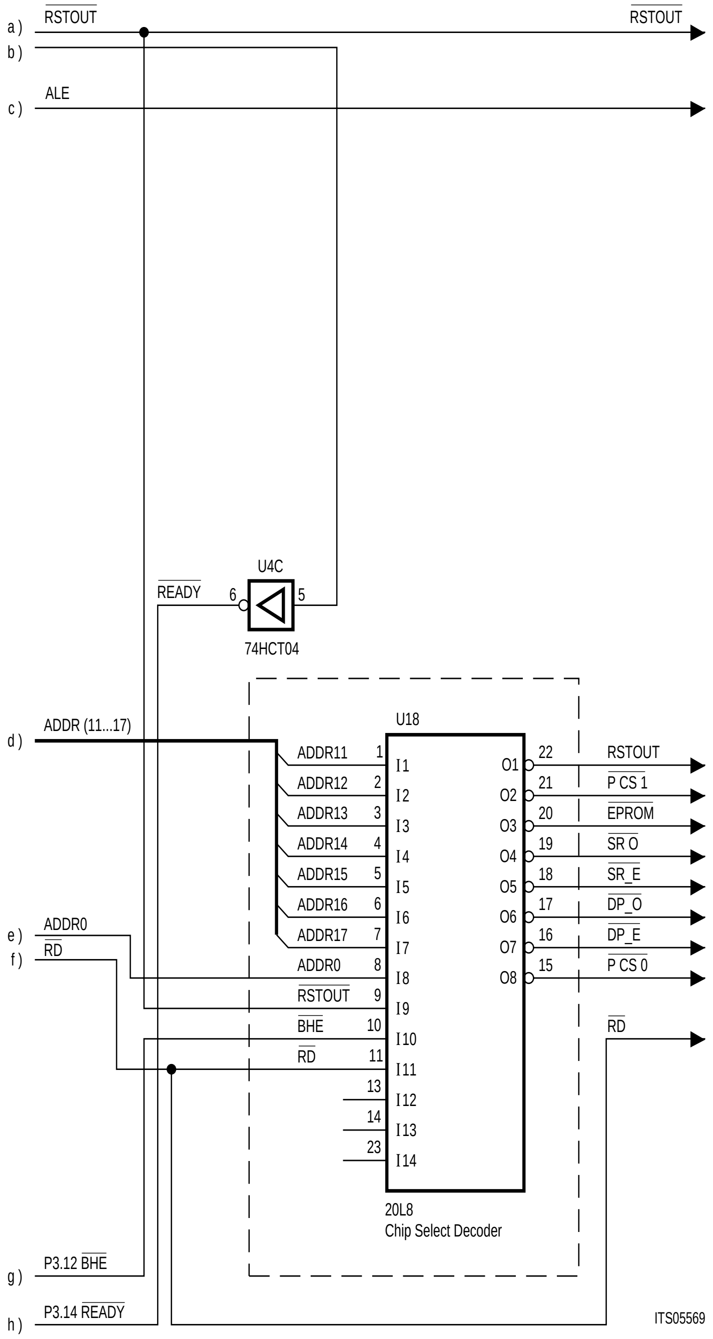
Dual Port RAM



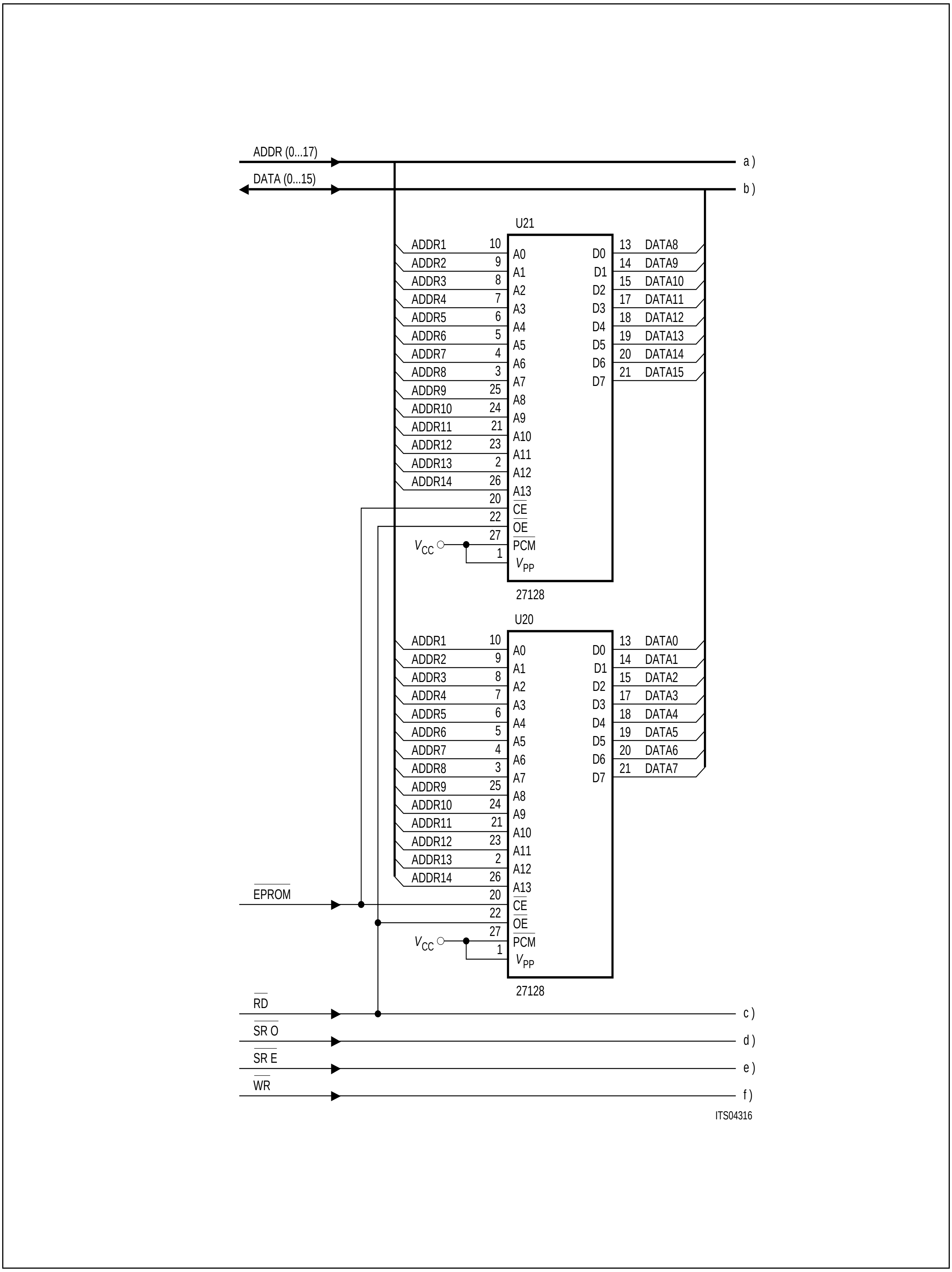
Dual Port RAM (cont'd)



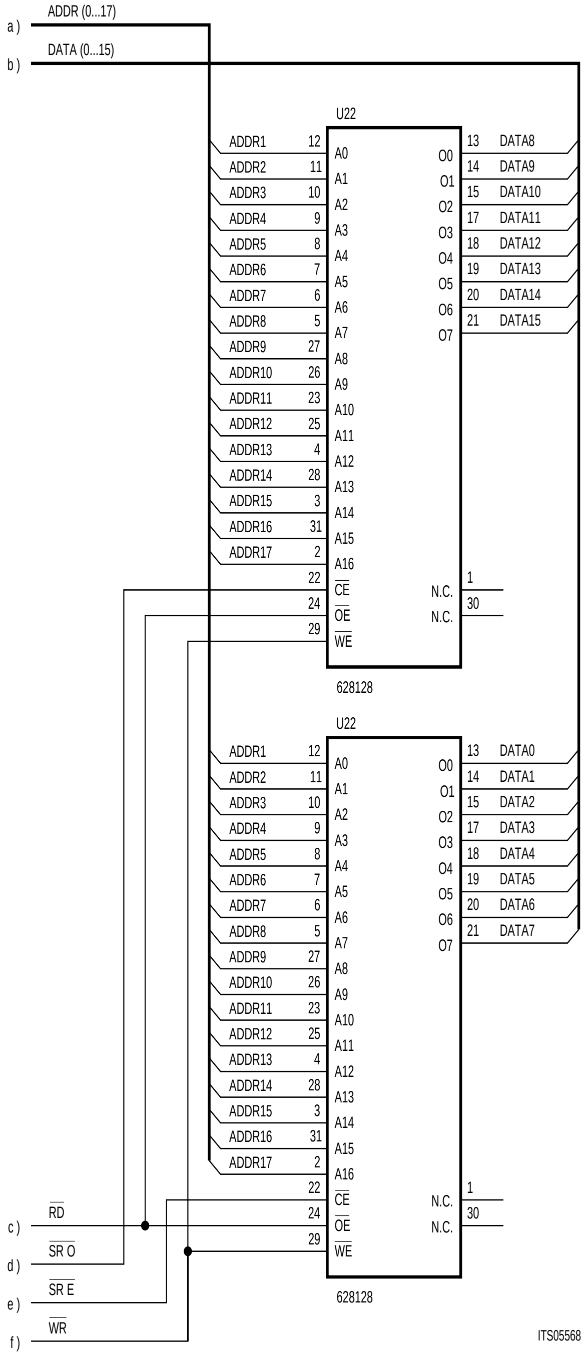
CPU-Block SAB 80C166



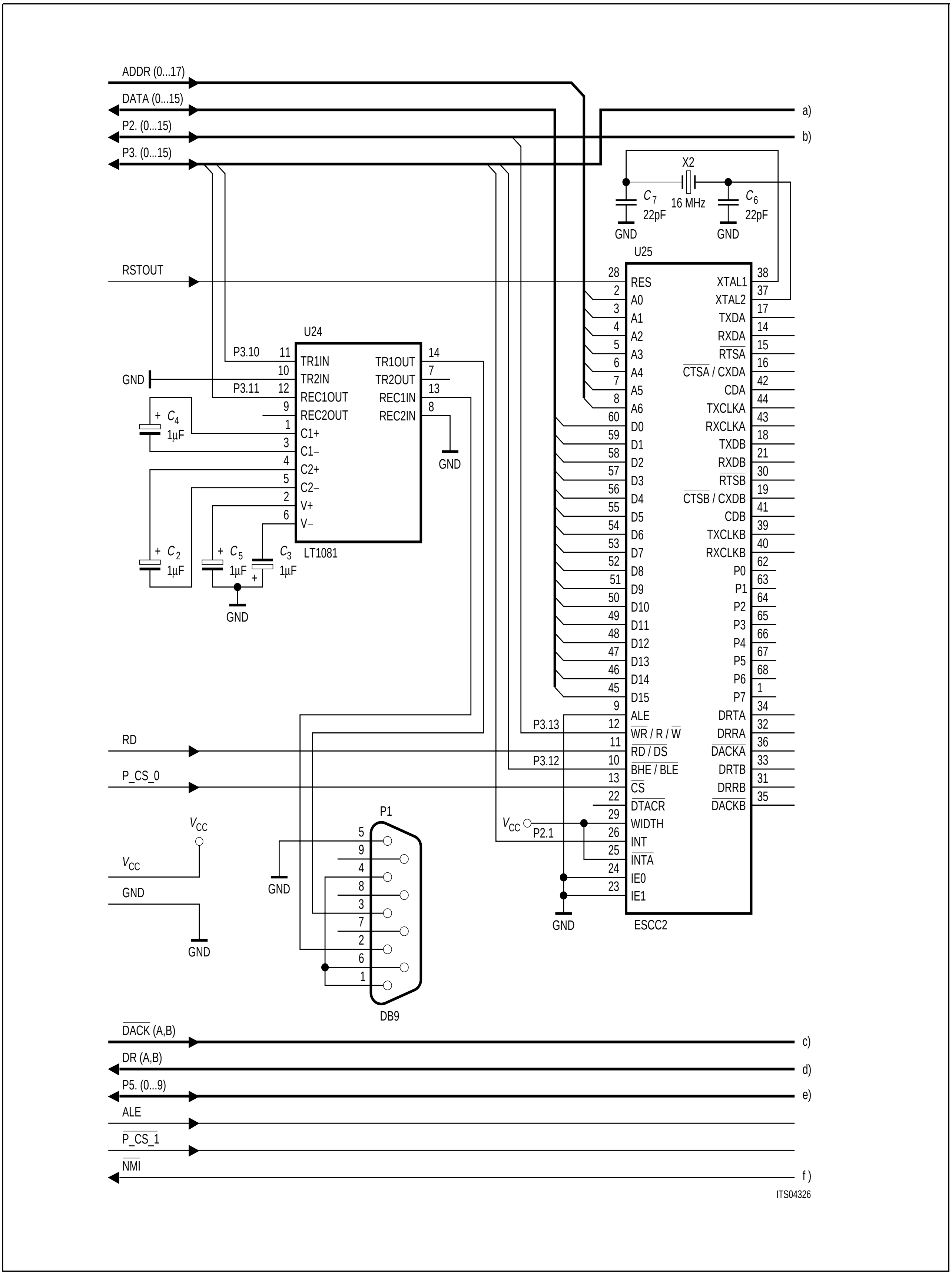
CPU-Block SAB 80C166 (cont'd)



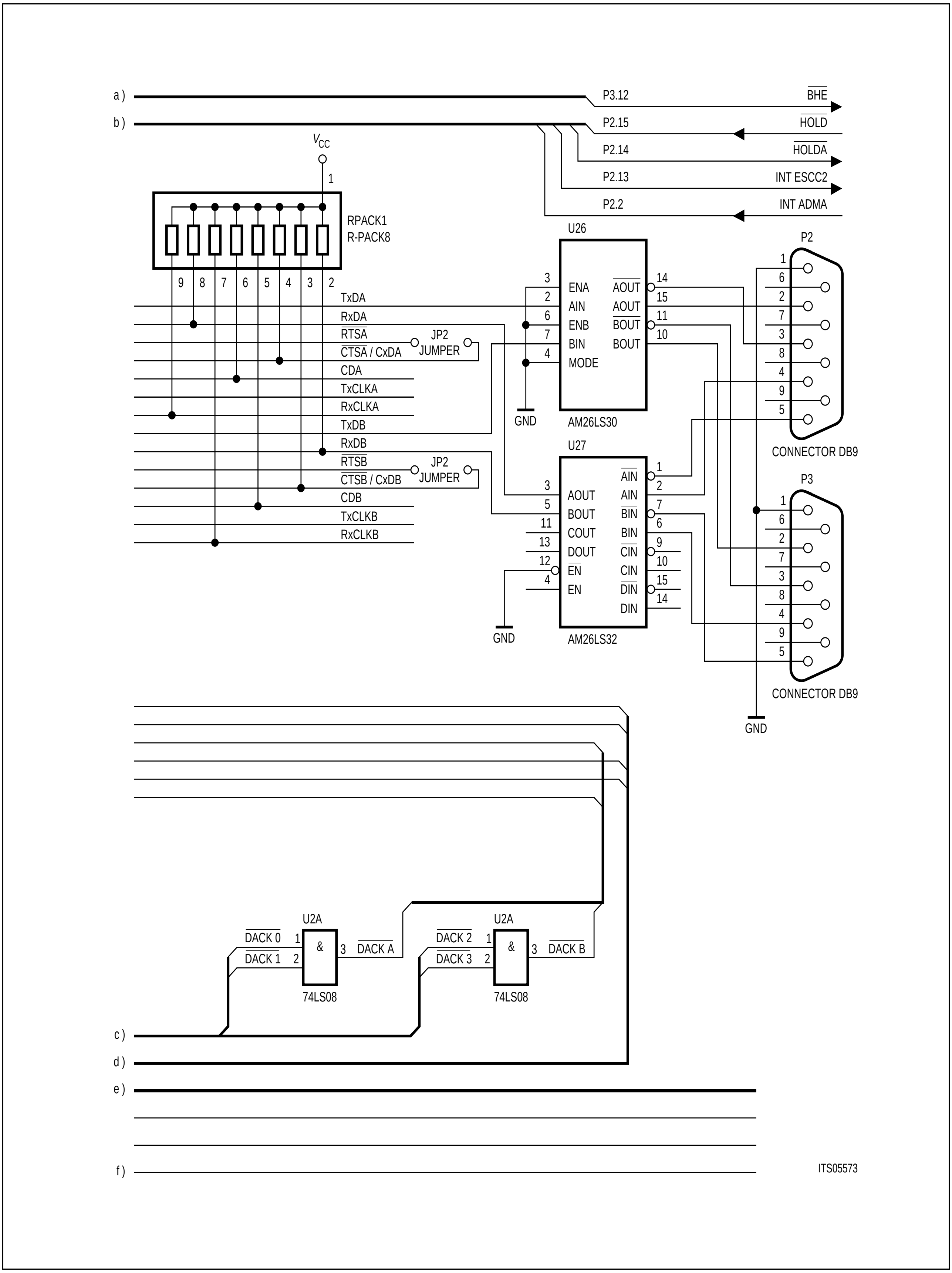
RAM and EPROM



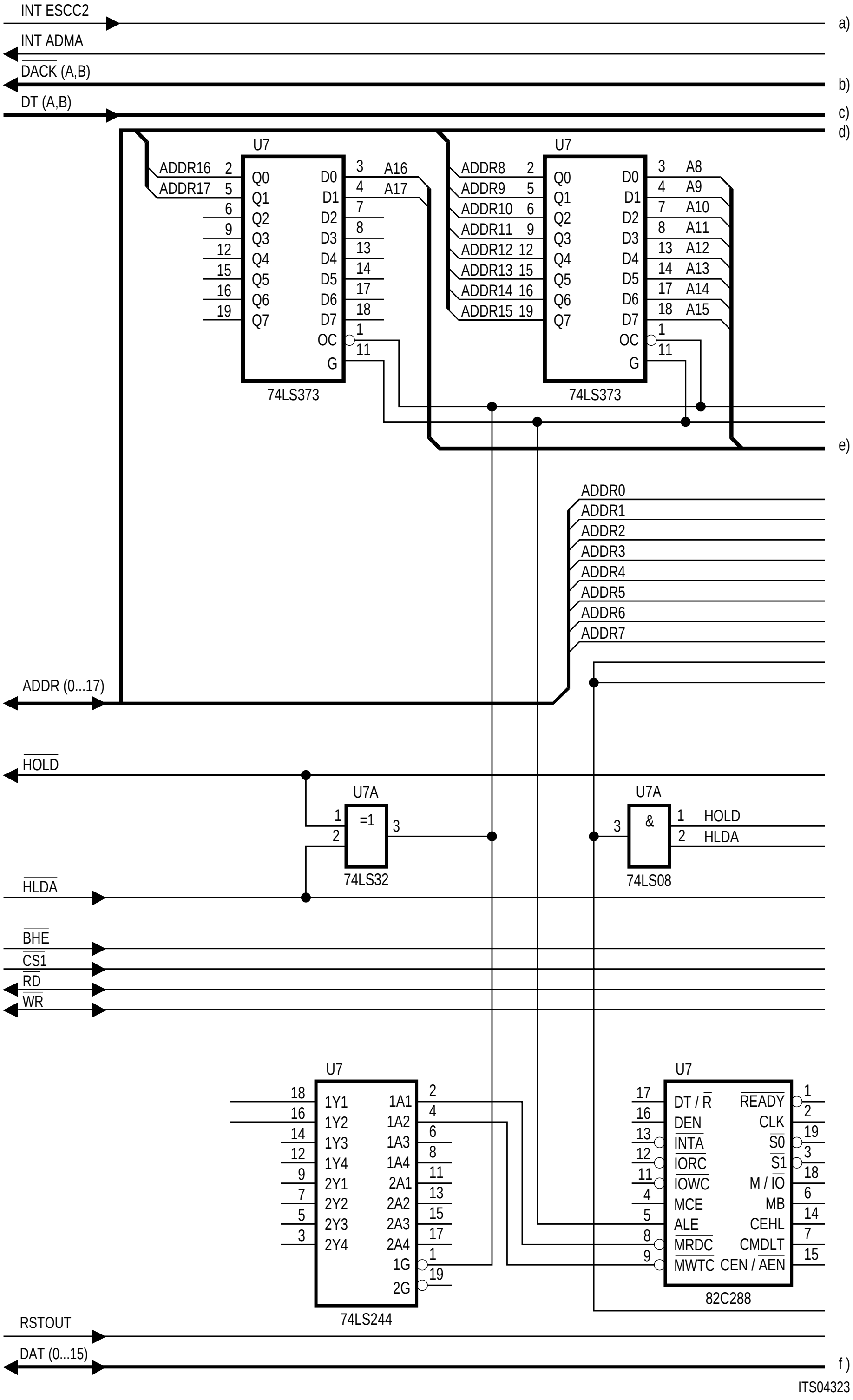
RAM and EPROM (cont'd)



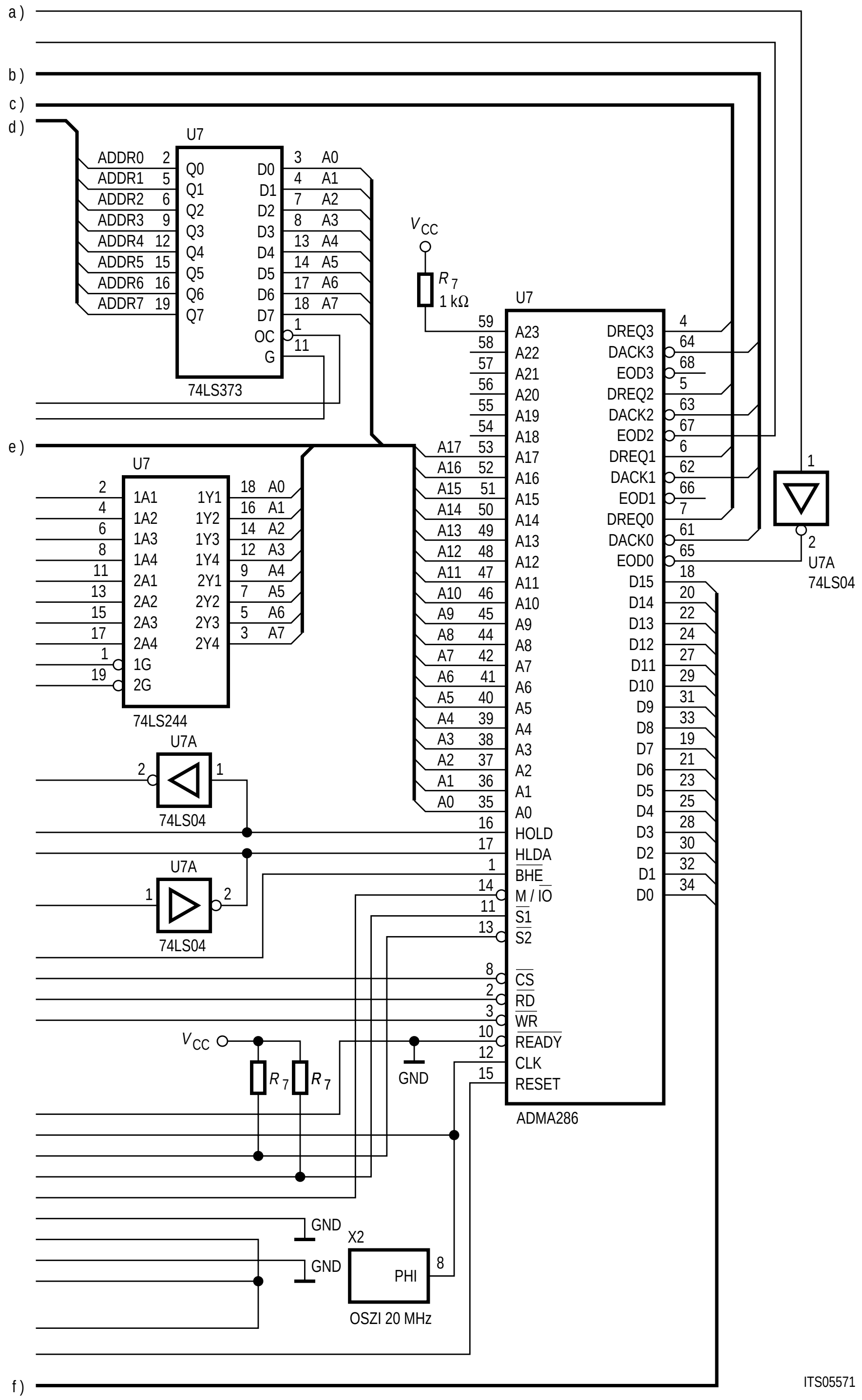
ESCC2-Serial Port



ESCC2-Serial Port (cont'd)


INT ESCC2
a)
INT ADMA
DACK (A,B)
b)
DT (A,B)
c)
d)
U7
ADDR16 2
ADDR17 5
6
9
12
15
16
19
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
D0
D1
D2
D3
D4
D5
D6
D7
OC
G
3 A16
4 A17
7
8
13
14
17
18
1
11
74LS373
U7
ADDR8 2
ADDR9 5
ADDR10 6
ADDR11 9
ADDR12 12
ADDR13 15
ADDR14 16
ADDR15 19
3 A8
4 A9
7 A10
8 A11
13 A12
14 A13
17 A14
18 A15
1
11
74LS373
e)
ADDR0
ADDR1
ADDR2
ADDR3
ADDR4
ADDR5
ADDR6
ADDR7
ADDR (0...17)
HOLD
U7A
1
2
=1
3
74LS32
U7A
3
&
1 HOLD
2 HLDA
74LS08
HLDA
BHE
CS1
RD
WR
U7
18 1Y1
16 1Y2
14 1Y3
12 1Y4
9 2Y1
7 2Y2
5 2Y3
3 2Y4
1A1 2
1A2 4
1A3 6
1A4 8
2A1 11
2A2 13
2A3 15
2A4 17
1G 1
2G 19
74LS244
U7
17 DT / R
16 DEN
13 INTA
12 IORC
11 IOWC
4 MCE
5 ALE
8 MRDC
9 MWTC
READY 1
CLK 2
S0 19
S1 3
M / IO 18
MB 6
CEHL 14
CMDLT 7
CEN / AEN 15
82C288
RSTOUT
DAT (0...15)
f)
ITS04323

ADMA


a)
b)
c)
d)
e)
f)
U7
ADDR0 2
ADDR1 5
ADDR2 6
ADDR3 9
ADDR4 12
ADDR5 15
ADDR6 16
ADDR7 19
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
D0
D1
D2
D3
D4
D5
D6
D7
OC
G
3 A0
4 A1
7 A2
8 A3
13 A4
14 A5
17 A6
18 A7
1
11
74LS373
VCC
R 7
1 kΩ
U7
2
4
6
8
11
13
15
17
1
19
1A1
1A2
1A3
1A4
2A1
2A2
2A3
2A4
1G
2G
1Y1
1Y2
1Y3
1Y4
2Y1
2Y2
2Y3
2Y4
18 A0
16 A1
14 A2
12 A3
9 A4
7 A5
5 A6
3 A7
74LS244
U7A
2
1
74LS04
U7A
1
2
74LS04
VCC
R 7
R 7
GND
GND
X2
GND
PHI
8
OSZI 20 MHz
U7
59
58
57
56
55
54
A17 53
A16 52
A15 51
A14 50
A13 49
A12 48
A11 47
A10 46
A9 45
A8 44
A7 42
A6 41
A5 40
A4 39
A3 38
A2 37
A1 36
A0 35
16
17
1
14
11
13
8
2
3
10
12
15
A23
A22
A21
A20
A19
A18
A17
A16
A15
A14
A13
A12
A11
A10
A9
A8
A7
A6
A5
A4
A3
A2
A1
A0
HOLD
HLDA
BHE
M / IO
S1
S2
CS
RD
WR
READY
CLK
RESET
DREQ3
DACK3
EOD3
DREQ2
DACK2
EOD2
DREQ1
DACK1
EOD1
DREQ0
DACK0
EOD0
D15
D14
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
4
64
68
5
63
67
6
62
66
7
61
65
18
20
22
24
27
29
31
33
19
21
23
25
28
30
32
34
ADMA286
1
2
U7A
74LS04
ITS05571

ADMA (cont’d)

8.2 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 evalution board.

CHIP MEM_DEC1 PALCE20V8

STRING EPROM_ADDR_SELECT ‘($\overline{A17}$ x $\overline{A16}$ x $\overline{A15}$)’
;00000H – 07FFFH

STRING DPRAM_ADDR_SELECT ‘($\overline{A17}$ x $\overline{A16}$ x A15 x A14 x A13 x $\overline{A12}$)’
;0E000H – 0EFFFH

STRING PERI_ADDR0_SELECT ‘($\overline{A17}$ x $\overline{A16}$ x A15 x A14 x A13 x A12 x $\overline{A11}$ x $\overline{A10}$)’
;0F000H – 0F3FFH

STRING PERI_ADDR1_SELECT ‘($\overline{A17}$ x $\overline{A16}$ x A15 x A14 x A13 x A12 x $\overline{A11}$ x $\overline{A10}$)’
;0F400H – 0F7FFH

EQUATIONS

EPROM_CS = EPROM_ADDR_SELECT x RD x $\overline{\text{RSTOUT}}$
PERI-CS_0 = PERI_ADDR0_SELECT
PERI-CS_1 = PERI_ADDR1_SELECT
DPRAM_CS_ODD = DPRAM_ADDR_SELECT x BHE
DPRAM_CS_EVEN = DPRAM_ADDR_SELECT x $\overline{\text{A0}}$
SRAM_CS_ODD = ($\overline{\text{EPROM}}$_ADDR_SELECT x ($\overline{\text{RSTOUT}}$+$\overline{\text{RSTOUT}}$ x $\overline{\text{RD}}$))
x $\overline{\text{PERI}}$_ADDR0_SELECT
x $\overline{\text{PERI}}$_ADDR1_SELECT
x $\overline{\text{DPRAM}}$_ADDR_SELECT
x BHE

SRAM_CS_EVEN = ($\overline{\text{EPROM}}$_ADDR_SELECT x ($\overline{\text{RSTOUT}}$+$\overline{\text{RSTOUT}}$ x $\overline{\text{RD}}$))
x $\overline{\text{PERI}}$_ADDR0_SELECT
x $\overline{\text{PERI}}$_ADDR1_SELECT
x $\overline{\text{DPRAM}}$_ADDR_SELECT
x $\overline{\text{A0}}$

8.3 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.