SIEMENS

ICs for Communications

Advanced ISDN Videophone Controller
SAB 83C515A-AVPC Version 2.0

User's Manual 08.96

Edition 06.96

This edition was realized using the software system FrameMaker®.

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

Attention please!

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.

Components used in life-support devices or systems must be expressly authorized for such purpose!

Critical components[1] of the Semiconductor Group of Siemens AG, may only be used in life-support devices or systems[2] with the express written approval of the Semiconductor Group of Siemens AG.

1 A critical component is a component used in a life-support device or system whose failure can reasonably be expected to cause the failure of that life-support device or system, or to affect its safety or effectiveness of that device or system.

2 Life support devices or systems are intended (a) to be implanted in the human body, or (b) to support and/or maintain and sustain human life. If they fail, it is reasonable to assume that the health of the user may be endangered.

SAB 83C515A-AVPC Revision History:		Current Version: 06.96
Previous Version:		1.1
Page (in previous Version)	Page (in current Version)	Subjects (major changes since last revision)
5-18	6-30	Description of SW structure and new interface of AVPC Version 2.0 (World Wide)
33-42	46-57	Changes of UCIF for AVPC Version 2.0 (World Wide)

Table of Contents **Page**

Table of Contents **Page**

1 Functional Description

1.1 Overview

The Advanced ISDN Videophone Controller SAB 83C515A-AVPC in combination with the Siemens ISDN Chip Set (JADE, ISAC-S TE, ARCOFI-SP) forms a perfect system solution for ISDN basic rate applications.

The SAB 83C515A-AVPC firmware contains a complete solution for five basic rate ISDN D channel signalling protocols. The firmware has passed conformance tests for the following D channel protocols:

- DSS1 (Europe) according to ETS 300 125, ETS 300 153 (Layer 2) and ETS 300 102, ETS 300 104 (Layer 3).
- DMS100 Custom (USA) according to Northern Telecom NIS S208-6 (Layer 2 & 3).
- 5ESS Custom (USA) according to AT&T 235-900-343 (Layer 2 & 3).
- NI-1 (USA) according to Bellcore TR-TSY-000793 (Layer 2) and SR-NWT-001953 (Layer 3).
- NTT (Japan) according to NTT INS-Net Interface and Services (Layer 2 & 3).

The firmware is accessed via easy-to-use system control primitives. As the D channel protocol is processed by AVPC, deterministic protocol response times are feasible even within multitasking systems.

Besides the D channel protocol control, the AVPC provides transparent ARCOFI-SP access for the host application (Speakerphone support). The interaction between the Siemens Chip Set hardware and the software is shown in **Figure 1**.

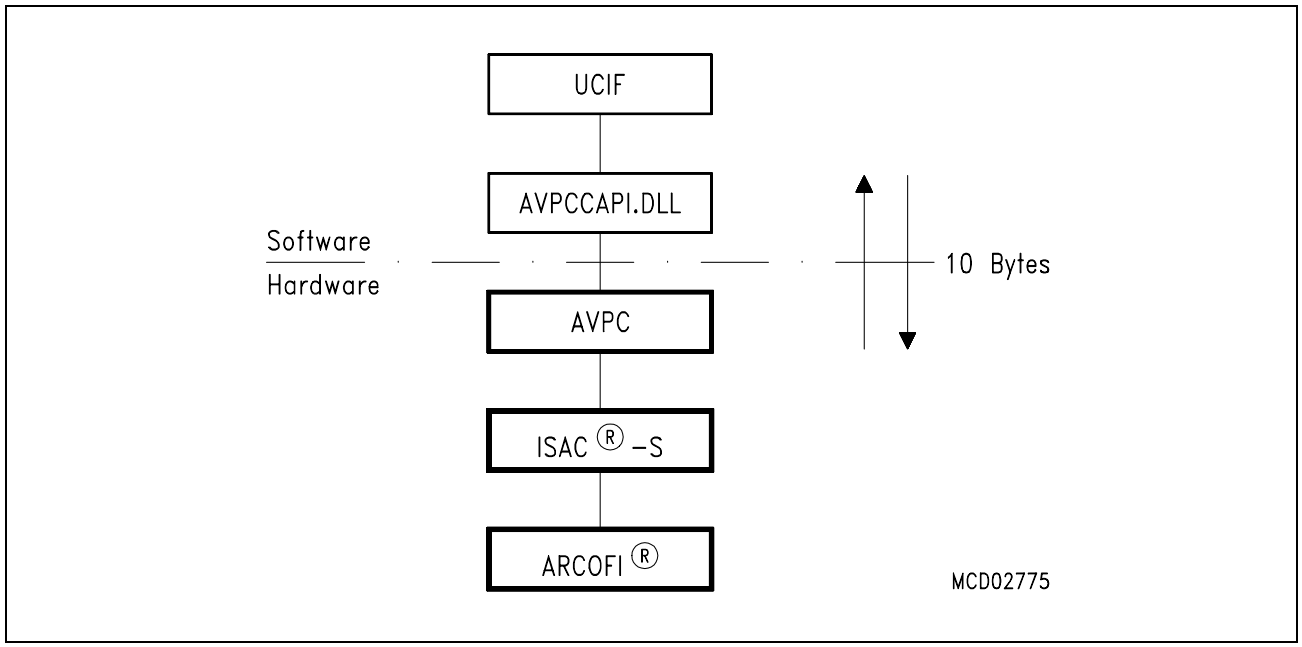

Figure 1
Interaction between Software and Hardware

1.2 Software Structure

The software structure of the AVPC can be divided in six major parts (**see Figure 2, 3 and 4**).

- The **Interface handler** is responsible for the message transfer from the host to the AVPC and vice versa. The handler decodes the commands received from the host application and passes the information to the corresponding software block. The Interface handler is also responsible for sending indications to the host, in order to inform the host about received call progress messages. A detailed description of all commands and indications (system control primitives) can be found in **Chapter 1.3**.
- The **BVC** process (Basic Voice Control) is placed on top of the AVPC protocol stack and represents the application layer. The process is responsible for converting received system control primitives in suitable network layer frames. Because the AVPC can handle two independent calls simultaneously, two instances of the BVC process are implemented. The 'call_id' parameter of a system control primitive is used to distinguish between the two instances. The BVC process routes the assembled network layer frame to the assigned underlying Q.931 process.
- For every BVC instance a **Q.931** process instance is implemented. This process represents the network layer protocol according to the implemented Layer 3 specifications. All network layer 3 messages from both Q.931 instances are passed on to the Q.921 protocol block.
- The **Q.921** protocol block provides a data link for the two Q.931 processes according to the implemented Layer 2 specifications. All network layer messages are routed over the same data link layer (if DSS1 or NTT protocol is set) or two independent data link layers (if NI-1, DMS-100 or 5ESS Custom is set). In all cases the Q.921 protocol block uses the full transparent mode feature of the ISAC-S. The Layer 3 messages can be distinguished by different call reference values (CR). If two data link layers are used they can be distinguished by different Terminal Endpoint Identifier (TEI).
- The **MMC** (Management Maintenance Control) process is responsible for the SPID (Service Profile Identifier) registration which is necessary for all USA protocols. With the SPID values, terminals acknowledge themselves to the local switch after Layer 2 is established. The MMC process is located at the top of the AVPC protocol next to the BVC process.
- The **IOM-2 handler** is responsible for the ARCOFI-SP programming. The data is written to the ARCOFI-SP using the MONITOR 1 channel of the IOM-2 interface. This allows a host application to program the ARCOFI-SP transparently (e.g speakerphone support).

The following three diagrams give an overview of the software structure inside the AVPC. The software structure depends on the chosen D channel protocol (switch type) and the number of implemented SPIDs. Grayed blocks in the diagrams are deactivated.

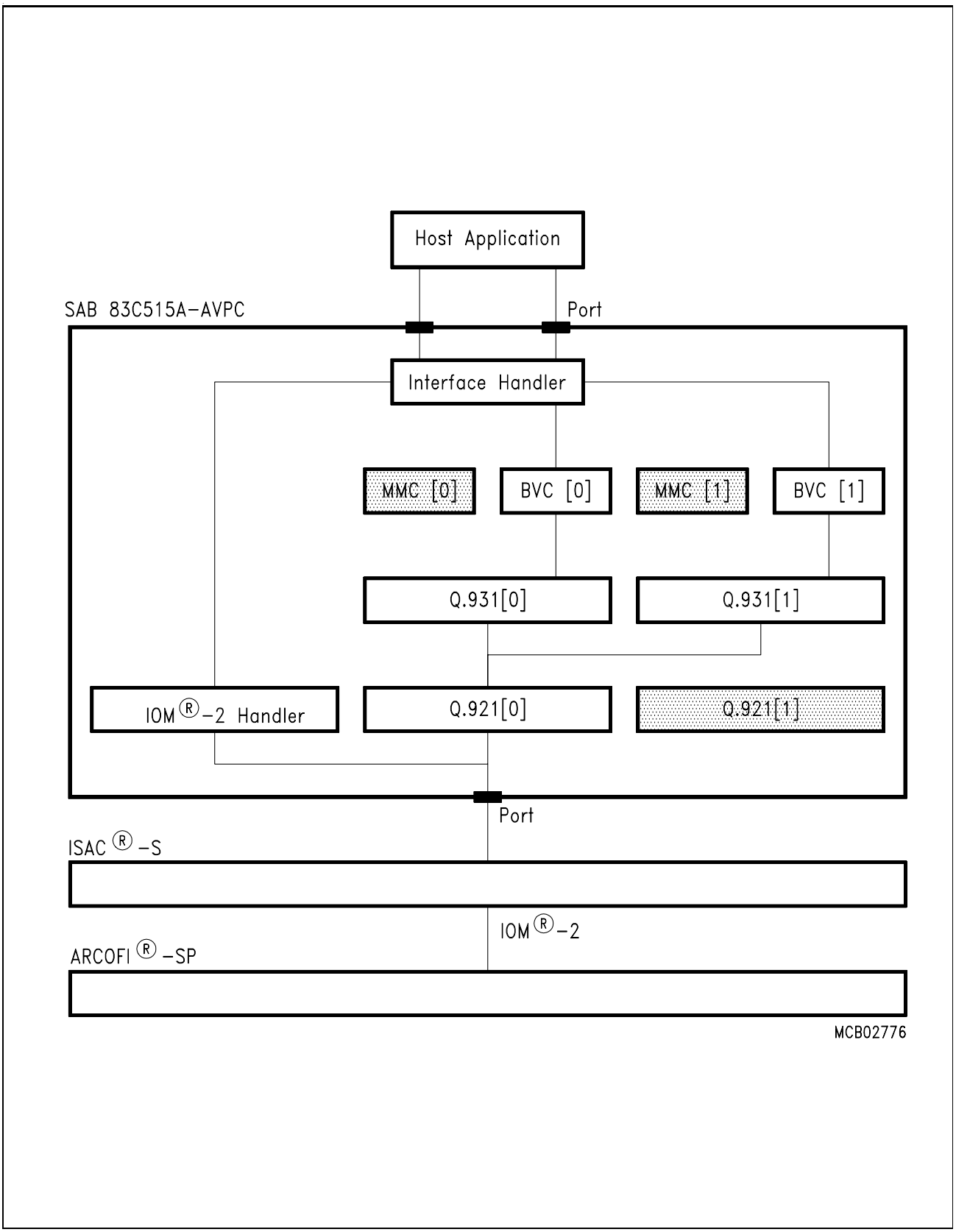

Figure 2
Block Diagram of the SAB 83C515A-AVPC (Switch type: DSS1 or NTT)

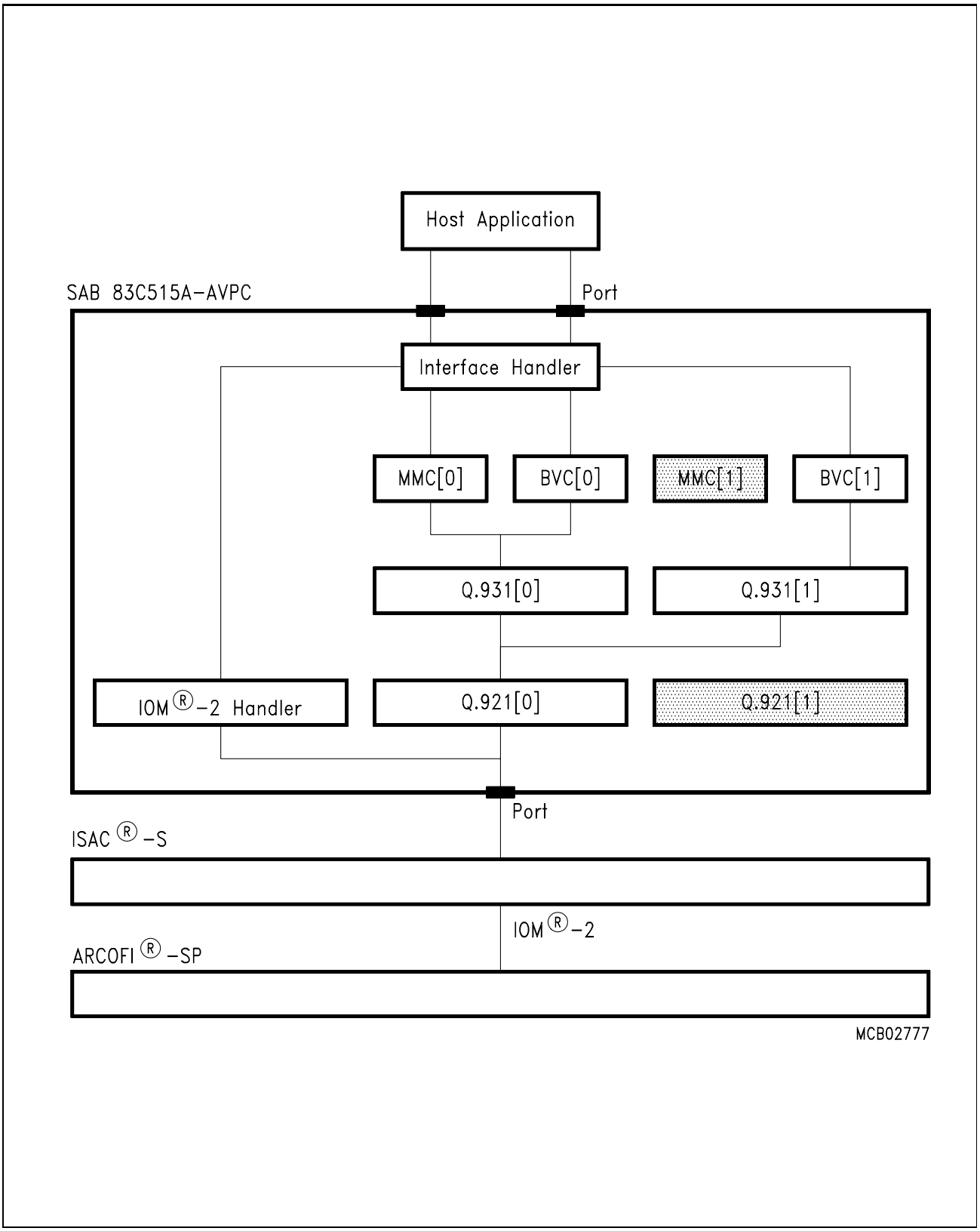

Figure 3
Block Diagram of the SAB 83C515A-AVPC (Switch type: 5ESS Custom, NI-1 or DMS-100, using one SPID)

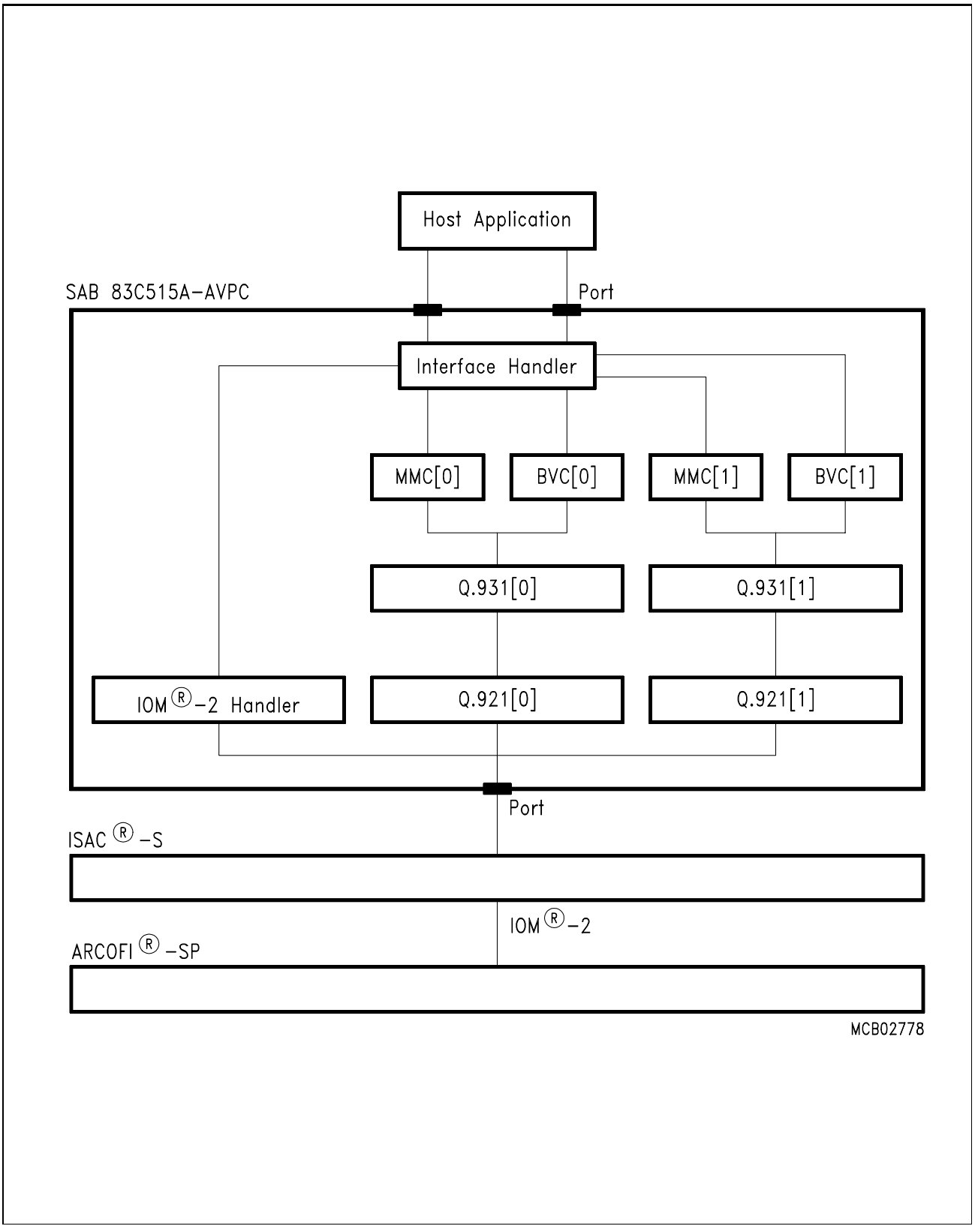

Figure 4
Block Diagram of the SAB 83C515A-AVPC (Switch type: 5ESS Custom, NI-1 or DMS-100, using two SPIDs)

1.3 System Control Primitives

The information exchange between the host application and the AVPC is performed via system control primitives. In order to simplify the information exchange, all system control primitives have a fixed length of 10 Bytes. The primitives can be divided into two main classes. Primitives which are transferred from the host to the AVPC are referenced in the following as commands. In the opposite direction these primitives are named indications. The command primitives can be subdivided into call control commands, ARCOFI-SP programming commands and ISAC-S register access command. The indication primitives can be divided in call control indications and error indications (**see Figure 5**).

Because the AVPC is capable of handling two independent calls simultaneously, it is necessary to specify the call control instance to which a primitive belongs. System control primitives with a *call_id* of 0x00 are routed by the first call control instance (BVC[0]), primitives with a *call_id* of 0x01 are routed by the second one (BVC[1]). All other values are not allowed.

Figure 5 gives an overview of the system control primitive organization of the AVPC.

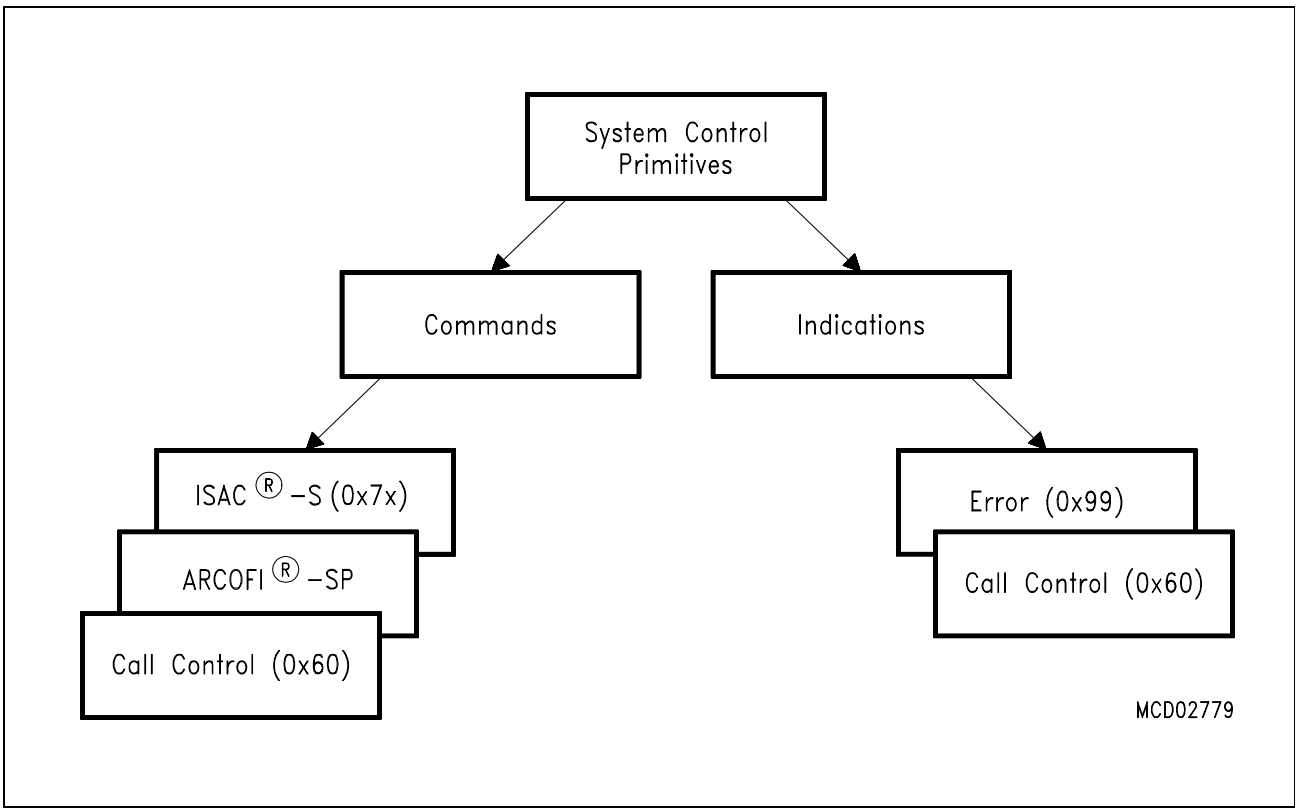

Figure 5
System Control Primitives of the AVPC

1.3.1 Call Control Commands

USER_CONN_RQ

This primitive is transferred to the AVPC whenever the host application wants to establish an ISDN call. If a call is initiated, AVPC sends the called party number via block dialing to the switch. With block dialing the whole number information is sent within one outgoing setup frame. Therefore it is necessary to hand over the complete called party number to AVPC (with KEYPAD_RQ) before initiating an outgoing call with USER_CONN_RQ.

0x60	0x10	call_id	c_type	b_ch	HLC	calling PNT	called PNT	send LN	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

c_type: 0x00 an ISDN voice call (uLaw) is requested.
0x01 an ISDN voice call (ALaw) is requested.
0x02 an ISDN data call (transp. 64 kbits) is requested.
0x03 an ISDN data call (transp. 56 kbits) is requested.

b_ch: 0x01 one B channel is requested.

HLC: High Layer Compatibility:
0x00 none.
0x01 Telephony.
0x04 Facsimile Group 2/3.
0x21 Facsimile Group 4 Class I.
0x24 Facsimile Group 4 Class II + III.

callingPNT: Number type which is included in the calling party number information element added to the outgoing setup frame. If it is unknown which value the switch expects use 0xC1 as default.

calledPNT: Number type which is included in the called party number information element added to the outgoing setup frame. If it is unknown which value the switch expects use 0x81 as default.

send LN: 0x00 disable ‘send local number’.
0x01 enable ‘send local number’: the calling party number information element is added to the outgoing setup.

X: doesn’t matter.

USER_CLEAR_RQ

The primitive is used to clear an ISDN call by the application program.

0x60	0x11	call_id	X	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

X: doesn't matter.

KEYPAD_RQ

The primitive is used to pass dialing information for an initiated ISDN call to the AVPC. Each call number character must be transmitted with one KEYPAD_RQ. The last digit must be completed with the terminator 0x00 (see example below).

0x60	0x12	call_id	digit	index	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

digit: dialing information (ASCII coded, 0x0X) or terminator (0x00).

index: the index into the dial number string, it ranges from 0 to n, the maximum index is 20.

X: doesn't matter.

Example: Sequence for transmitting the call number 399 to the AVPC:
0x60 0x12 call_id 0x33 0x00 X X X X X
0x60 0x12 call_id 0x39 0x01 X X X X X
0x60 0x12 call_id 0x39 0x02 X X X X X
0x60 0x12 call_id 0x00 0x03 X X X X X

SPID_ASSIGN_RQ

The primitive is used to pass SPID information to the AVPC. This call is equivalent to KEYPAD_RQ with one exception: a SPID is transmitted instead of a phone number.

0x60	0x20	call_id	digit	index	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

digit: SPID information (ASCII coded, 0x0X) or terminator (0x00).

index: the index into the dial number string, it ranges from 0 to n, the maximum index is 20.

X: doesn't matter.

ASSIGN_SWITCH_TYPE_RQ

The primitive is used to pass the type of the switch to the AVPC. This must be the first primitive sent if AVPC is in the resetted state. As long as the switch type is not assigned all other commands will be ignored by AVPC. After the assignment, the available SPIDs and local phone numbers must be transmitted. The switch type can only be selected once. Subsequent ASSIGN_SWITCH_TYPE_RQ calls will be ignored. If the switch type has to be reselected it is necessary to reset the AVPC before (with reset program RESETUC.PIF and RESETUC.EXE).
The AVPC answers every ASSIGN_SWITCH_TYPE_RQ with a STATUS_IN which contains the currently selected switch type along with other information.

0x60	0x21	0x00	type	X	X	X	X	X	X

type: 0x01 DSS1 (Europe).
0x02 NTT (Japan).
0x04 5ESS Custom (USA).
0x08 NI-1 (USA).
0x10 DMS-100 (USA).

X: doesn't matter.

GET_STATUS_RQ

The primitive is used to request a status message from the AVPC (see STATUS_IN). Additionally this message is used as a control message for the watchdog timer (see ENABLE_WATCHDOG_TIMER_RQ).

0x60	0x22	X	X	X	X	X	X	X	X

X: doesn't matter.

ASSIGN_MSN_RQ

The primitive is used to transfer a Multiple Subscriber Number (MSN) to the AVPC. The transferred MSN is equivalent to the local phone number and must be entered without an area code. The MSN has to be transmitted byte by byte to the AVPC with ASSIGN_MSN_RQ completed by a terminator (see also KEYPAD_RQ). AVPC will only accept incoming calls for the assigned MSNs.

0x60	0x23	call_id	digit	index	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

digit: MSN information (ASCII coded, 0x0X) or terminator (0x00).

index: the index into the dial number string, it ranges from 0 to n, the maximum index is 20.

X: doesn't matter.

ENABLE_WATCHDOG_TIMER_RQ

The primitive is used to control the watchdog timer. By default watchdog timer is switched on. If watchdog timer is enabled AVPC is waiting for a GET_STATUS_RQ cyclical sent from the host application. If AVPC doesn't receive this message at least every 60 sec all outgoing lines will be disconnected and AVPC has to be resetted (with reset program RESETUC.PIF and RESETUC.EXE).

0x60	0x24	enable	X	X	X	X	X	X	X

enable: 0x00 disables the usage of the watchdog timer.
0x01 enables the usage of the watchdog timer.

X: doesn't matter.

ENABLE_LISTEN_RQ

The primitive is used to enable AVPC listening to incoming data and/or voice calls. By default AVPC listens to both incoming data and voice calls.

0x60	0x27	call_id	enable	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

enable: 0x00 disables listen to incoming data and voice calls.
0x01 enables listen to incoming voice calls.
0x02 enables listen to incoming data calls.
0x03 enables listen to incoming data and voice calls.

X: doesn't matter.

USER_ACCEPT_RQ

This primitive is used by the host application to accept an incoming call.

0x60	0x28	call_id	X	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

X: doesn't matter.

1.3.2 ARCOFI®-SP Commands

The host application uses these commands to program ARCOFI-SP via AVPC. The IOM-Handler of the AVPC writes the received data string transparently to the ARCOFI-SP (IOM2-Address 0xA1) using the MONITOR1 channel of the IOM2-Interface. The host application thus controls directly the behavior of the ARCOFI-SP (switching function, codec function, tone generation, speakerphone function).

Note: The AVPC firmware does not do any ARCOFI-SP programming! All programming must be done by the host application using the ARCOFI-SP commands.

ARC_1COEF_RQ

The primitive is used to pass a one byte ARCOFI-SP sequence to the AVPC.

0x71	P	X	X	X	X	X	X	X	X

P: ARCOFI-SP programming sequence.
X: doesn’t matter.

ARC_2COEF_RQ

The primitive is used to pass a two byte ARCOFI-SP sequence to the AVPC.

0x72	P	P	X	X	X	X	X	X	X

P: ARCOFI-SP programming sequence.
X: doesn’t matter.

ARC_3COEF_RQ

The primitive is used to pass a three byte ARCOFI-SP sequence to the AVPC.

0x73	P	P	P	X	X	X	X	X	X

P: ARCOFI-SP programming sequence.
X: doesn’t matter.

ARC_5COEF_RQ

The primitive is used to pass a five byte ARCOFI-SP sequence to the AVPC.

0x75	P	P	P	P	P	X	X	X	X

P: ARCOFI-SP programming sequence.
X: doesn’t matter.

ARC_9COEF_RQ

The primitive is used to pass a nine byte ARCOFI-SP sequence to the AVPC.

0x79	P	P	P	P	P	P	P	P	P

P: ARCOFI-SP programming sequence.

1.3.3 ISAC®-S Command

This command allows the host application to set a specified register of the ISAC-S to the given value. The register access of the ISAC-S is necessary for Layer-1 test support (e.g. B-Loop switching).

0x80	H.-Ad.	L.-Ad.	value	X	X	X	X	X	X

H.-Ad.: ISAC-S register address (high byte).
L.-Ad.: ISAC-S register address (low byte).
value: ISAC-S register value.
X: doesn’t matter.

1.3.4 Call Control Indications

USER_CONN_IN

The primitive indicates the receipt of a CONNECT_ACK network layer message by AVPC. This is the ‘normal’ response to the USER_CONN_RQ command primitive when the host application accepts an incoming call.

0x60	0x10	call_id	X	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.
X: doesn’t matter.

USER_CLEAR_IN

The USER_CLEAR_IN primitive is sent from the AVPC to the host application as a normal response to a previous USER_CLEAR_RQ. The indication is used to inform the host application that the other party has cleared the call. Additionally, this primitive is sent to the host application whenever a call establishment request is refused by the switching network. Causes with values 0x0? represent internal errors all other values concern external errors from the switch.

0x60	0x11	call_id	cause	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.
cause: 0x00 unspecified.
0x01 SPID not initialized, an SPID_ASSIGN_RQ is needed.
0x02 No call reference available.
0x03 No phone number specified.
0x04 The switch has assigned a B-channel that is wrong or already in use.
0x05 No answer to SETUP.
0x06 Lost of the layer 2 connection.
0x07 Timeout of timer T309 (layer 2 lost).
0x08 The primitive from the host application is not compatible to the current call state.
0x09 Wrong SPID number assigned. Host application has to be restarted

(only NI-1and DMS-100 Custom).

0x81 Unallocated (unassigned) number.

0x82 No route to specific transit network.

0x83 No route to destination.

0x86 Channel unacceptable.

0x87 Call awarded and being delivered in an established call.

0x90 Normal call clearing.

0x91 User busy.

0x92 No user responding.

0x93 No answer from user (user alerted).

0x95 Call rejected.

0x96 Number changed.

0x9A Non-selected user clearing.

0x9B Destination out of order.

0x9C Invalid number format (incomplete number).

0x9D Facility rejected.

0x9E Response to Status Enquiry.

0x9F Cause normal, unspecified.

0xA2 No circuit/channel available.

0xA3 Call queued.

0xA6 Network out of order.

0xA9 Temporary failure.

0xAA Switching equipment congestion.

0xAB Access information discarded.

0xAC Requested circuit/channel not available.

0xAF Resources unavailable, unspecified.

0xB1 Quality of service unavailable.

0xB2 Requested facility not subscribed.

0xB4 Outgoing calls barred.

0xB6 Incoming calls barred.

0xB9 Bearer capability not authorized.

0xBA Bearer capability not presently available.

0xBF Service or option not available, unspecified.

0xC1 Bearer capability not implemented.

0xC2 Channel type not implemented.
0xC5 Requested facility not implemented.
0xC6 Only restricted digital information bearer capability is available.
0xCF Service or option not implemented, unspecified.
0xD1 Invalid call reference value.
0xD2 Identified channel does not exist.
0xD3 A suspended call exists, but this call identity does not.
0xD4 Call identity in use.
0xD5 No call suspended.
0xD6 Call having the requested call identity has been cleared.
0xD8 Incompatible destination.
0xDB Invalid transit network selection.
0xDF Invalid message, unspecified.
0xE0 Mandatory information element is missing.
0xE1 Message type nonexistent or not implemented.
0xE2 Message not compatible with call state.
0xE3 Information element non-existent or not implemented.
0xE4 Invalid information element contents.
0xE5 Message not compatible with call state.
0xE6 Recovery on timer expiry.
0xEF Protocol error, unspecified.
0xFF Interworking, unspecified.

X: doesn’t matter

USER_B_CH_IN

The AVPC uses this primitive to report an incoming call to the host application. The call type and the specified B channel is included in the message. Additionally this primitive is used to indicate the successful B channel assignment by the switching network as a normal response to a previous call establishment request of the host application.

0x60	0x12	call_id	c_type	b_ch	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

c_type: 0x00 an ISDN voice call (uLaw) is indicated.
0x01 an ISDN voice call (ALaw) is indicated.
0x02 an ISDN data call (transp. 64 kbit) is indicated.
0x03 an ISDN data call (transp. 56 kbit) is indicated.
0xFF no information about the call type is available, when the indication is used as a response to a previous call establishment request.

b_ch: 0x00 B1 channel is assigned by the switching network.
0x01 B2 channel is assigned by the switching network.

X: doesn’t matter.

USER_CALL_PROCEEDING_IN

With this indication AVPC reports the receipt of a CALL PROCEEDING network layer message to the host application.

0x60	0x13	call_id	X	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.

X: doesn’t matter.

USER_ALERTING_IN

With this indication AVPC reports the receipt of an ALERTING network layer message to the host application.

0x60	0x14	call_id	X	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.
X: doesn't matter.

USER_CONNECT_IN

With this indication AVPC reports the receipt of a CONNECT network layer message to the host application.

0x60	0x15	call_id	X	X	X	X	X	X	X

call_id: 0x00 first call control instance.
0x01 second call control instance.
X: doesn't matter.

STATUS_IN

With this indication AVPC reports various status values. AVPC sends this indication after the receipt of ASSIGN_SWITCH_TYPE_RQ or GET_STATUS_RQ.

0x60	0x22	s_type	l3.1	l3.2	major	minor	X	X	X

s_type: chosen switch type (see ASSIGN_SWITCH_TYPE_RQ).
l3.1: Layer 3 call state (1st call instance).
l3.2: Layer 3 call state (2nd call instance).
major: the major AVPC release number.
minor: the minor AVPC release number.
X: doesn't matter.

1.3.5 Error Indication

With this indication AVPC reports the occurrence of an unrecoverable error to the host application. The AVPC must be resetted afterwards by the host application.

0x99	err_typ	X	X	X	X	X	X	X	X

err_typ:
- 0 internal Task Queue overflow.
- 2 Memory underrun.
- 5 Layer 3 Queue overflow.
- 6 Transmit data underrun reported by ISAC-S (interrupt EXIR:XDU).
- 7 Frame overflow reported by ISAC-S (interrupt EXIR:RFO).
- 9 Output Queue overflow.
- 10 Data overflow reported by the ISAC-S (interrupt EXIR:RDO).
- 11 IOM2 fatal error.

X: doesn’t matter.

1.4 Sample Message Flow

1.4.1 Message Flow for an Outgoing Call

The following figure shows the message flow between the AVPC and a host application for an outgoing call. The call is cleared by the host application again. In addition, the corresponding D channel messages (network layer) are shown in the right hand column. The parameters of the exchanged System Control Primitives are given in hexadecimal notation. Parameters that have no effect are not listed.

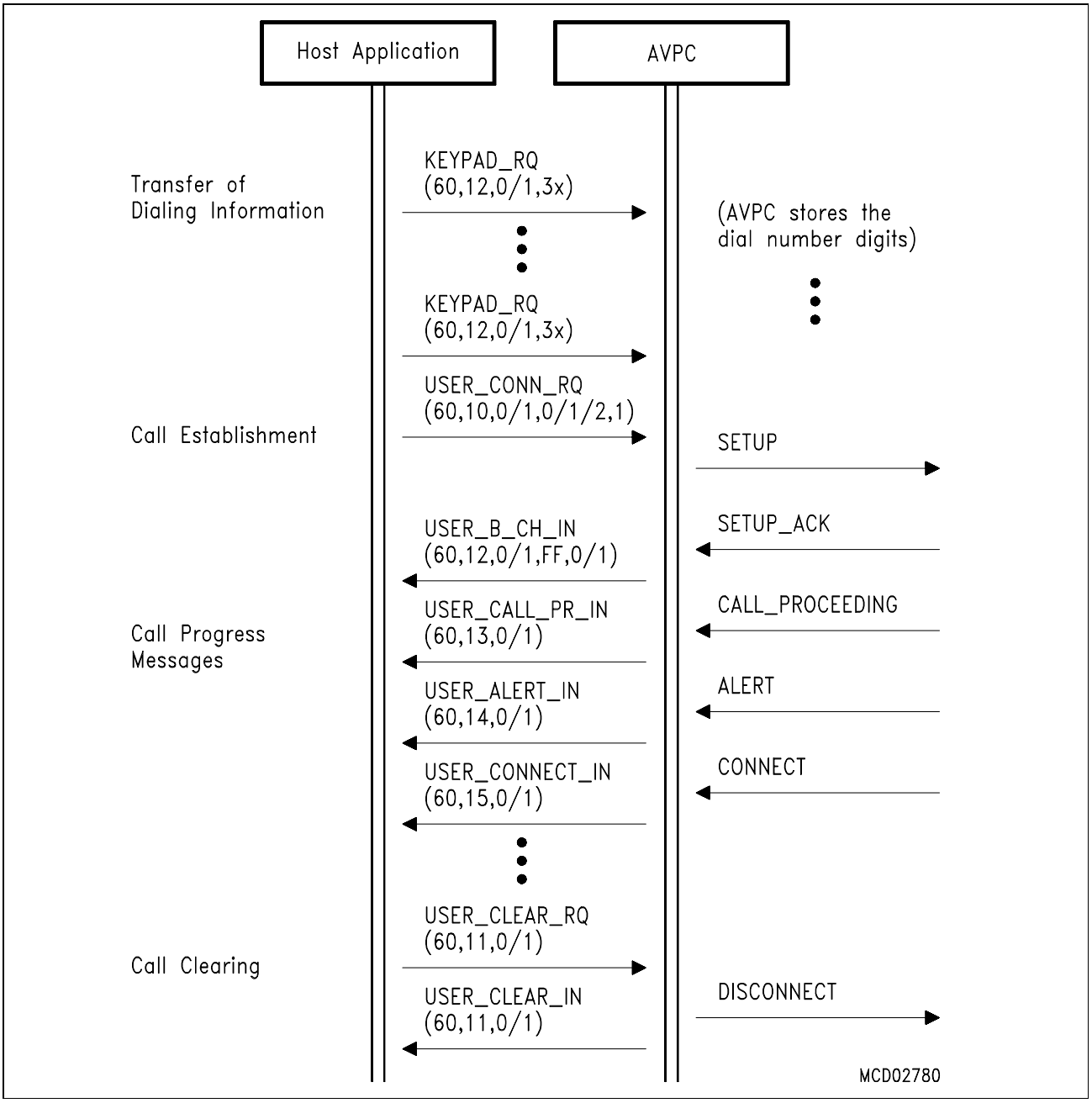

Figure 6
Message Flow for an Outgoing Call

1.4.2 Message Flow for an Incoming Call

Figure 7 demonstrates the message flow for an incoming call. The transferred messages between the AVPC and the host application are shown. The corresponding messages from the switch can be found in the right hand column. In this example the call is cleared by the calling party.

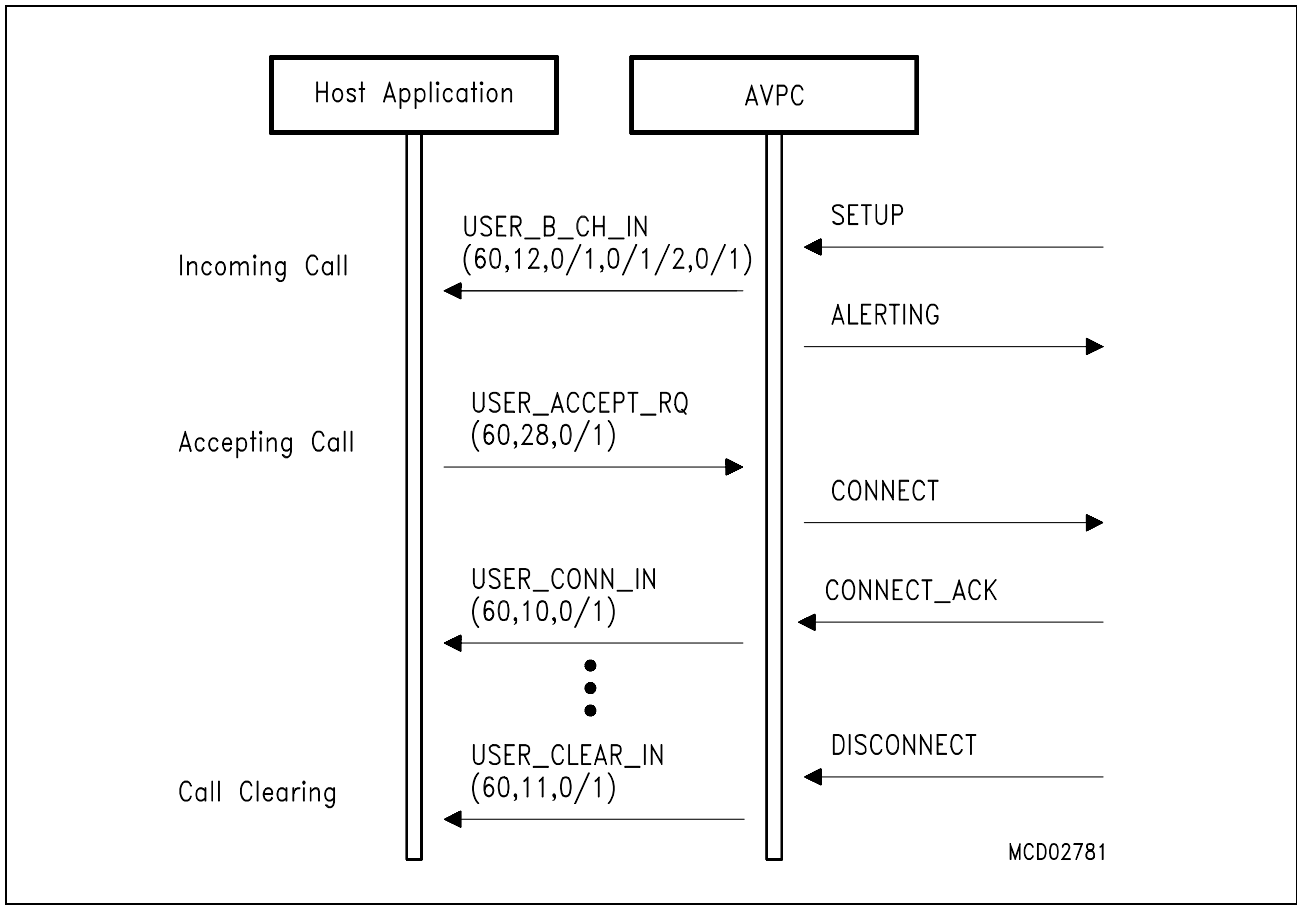

Figure 7
Message Flow for an Incoming Call

1.5 ARCOFI®-SP Programming

The AVPC provides the host application a transparent ARCOFI-SP access. The host application can use one of the five ARCOFI-SP programming commands to control the codec. It is important to know that the protocol part of the AVPC firmware does not do any ARCOFI-SP programming. All programming must be done by the host application (switching function, tone generation, speakerphone support).

1.6 Physical Layout

The following block diagram (**Figure 8**) shows the integration of the AVPC in a ISDN basic rate application. The application is divided into four parts:

- Host
- Interface
- AVPC
- ISAC-S and ARCOFI-SP

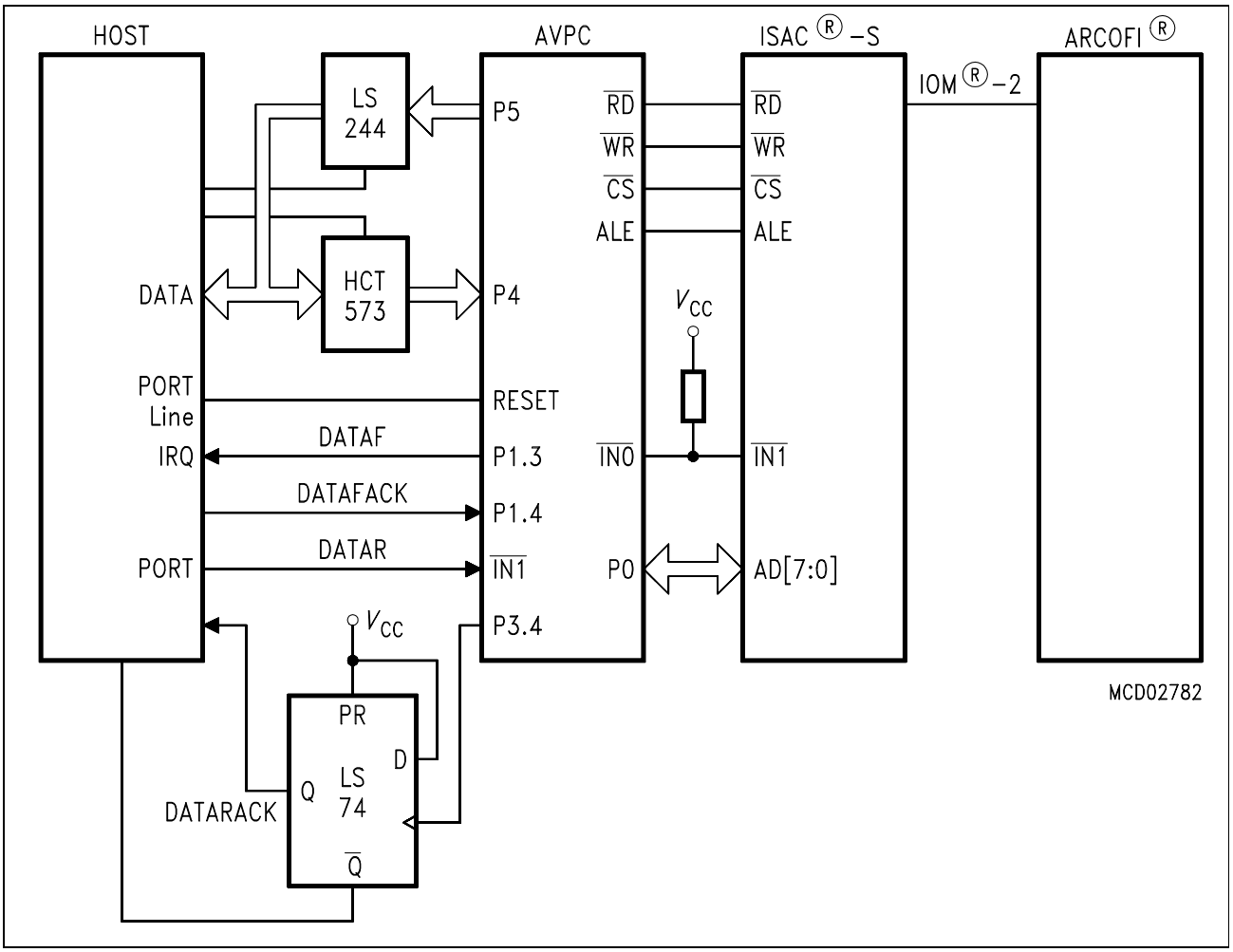

Figure 8
Hardware Block Diagram

The interface handles the data from Host to AVPC and vice versa. It is build with a ’LS244 Buffer and a ’HCT573 Latch. The four lines DATAF, DATAFACK, DATAR and DATARACK control the data flow via ’LS244 and ’HCT573.

Before any Control Primitive or Indication Primitive can be handled, the application software must set the DATAR line (DATAR=high) and the DATAFACK line (DATAFACK=high). Further the AVPC has to be resetted with a high pulse at the specific line and the DATAF line is set at the same time (DATAF=high). The AVPC then will generate a clock pulse at PORT 3.4 in order to set the DATARACK signal (DATARACK=high) of the ’LS74 (**see Figure 9**).

1.6.1 Send Command Primitives

The Host writes every byte of the Command Primitive to the ’HCT573 Latch. Writing to this latch resets the ’LS74 latch (DATARACK=low). The Host indicates that a byte is available with a high-low-high pulse at the DATAR line. Since the AVPC receives an Interrupt with the falling edge of DATAR, the interrupt will be serviced with a read of the latch data at PORT P4. The AVPC then sends a clock pulse at PORT P3.4 to set DATARACK (DATARACK=high). This indicates the Host that the written data is acknowledged and the next data can be sent. This procedure is continued until the 10 bytes of the Command Primitive are sent (**see Figure 9**).

1.6.2 Receive Indication Primitives

The AVPC presents the Indication Primitive byte by byte at PORT P5. The AVPC indicates that a byte is available by resetting the DATAF line (DATAF=low). So the HOST should receive an Interrupt with the falling edge of DATAF, the interrupt will be serviced with a read of buffered PORT P5 data. The ’LS244 Buffer isolates PORT P5 from the Host Data Bus until the data is read. The Host then sends a high-low-high pulse at DATAFACK line. This indicates the AVPC that the written data is acknowledged and the AVPC can send the next data. This procedure is continued until the 10 bytes of the Indication Primitive are received (**see Figure 9**).

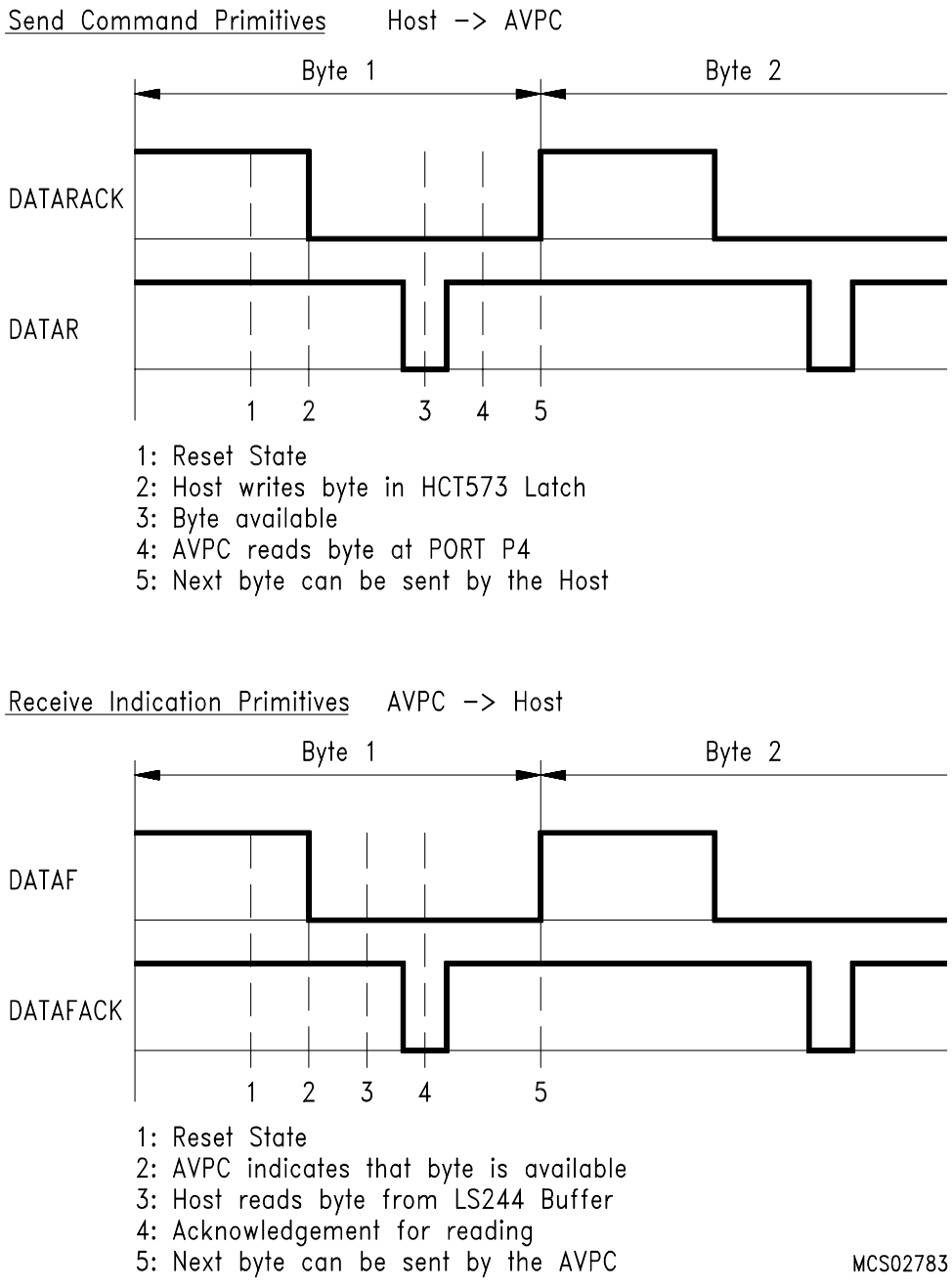

Figure 9
Interacting of the Signals between AVPC and Host

1.7 Special Notes

All indications are transferred byte by byte via an interrupt-driven handshake mechanism from the AVPC to the host application. The protocol firmware waits for confirmation from the host application for every byte AVPC sends. If the host application is not running (e.g. the Windows program has a system crash) the AVPC will hang in an infinite loop. As a result, the ISDN layers 2 and 3 can no longer be accessed and the current connection will be interrupted. Consequently, the host application should not be terminated as long as there are still active ISDN connections. The AVPC should be resetted whenever the host application is started and terminated.

2 Using AVPC for Videophone Application (DVC5-Board)

As described in **Chapter 1**, the AVPC forms a perfect system solution for ISDN basic rate applications in combination with the Siemens ISDN Chip Set (JADE, ISAC-S TE, ARCOFI-SP). One possible application is proposed with the DVC5-Board, which can be used as a videophone. This videophone application is available as: ‘Video Reference Board (DVC5-Board) Package SIPB 7280 V1.1’. In this package the necessary source codes are included.

The block diagram of the DVC5-Board and the flow of software control which is necessary is shown in **Figure 10**. Because the VPIC is the interface of the board to the PCI bus it is involved in all of the three different flows of control.

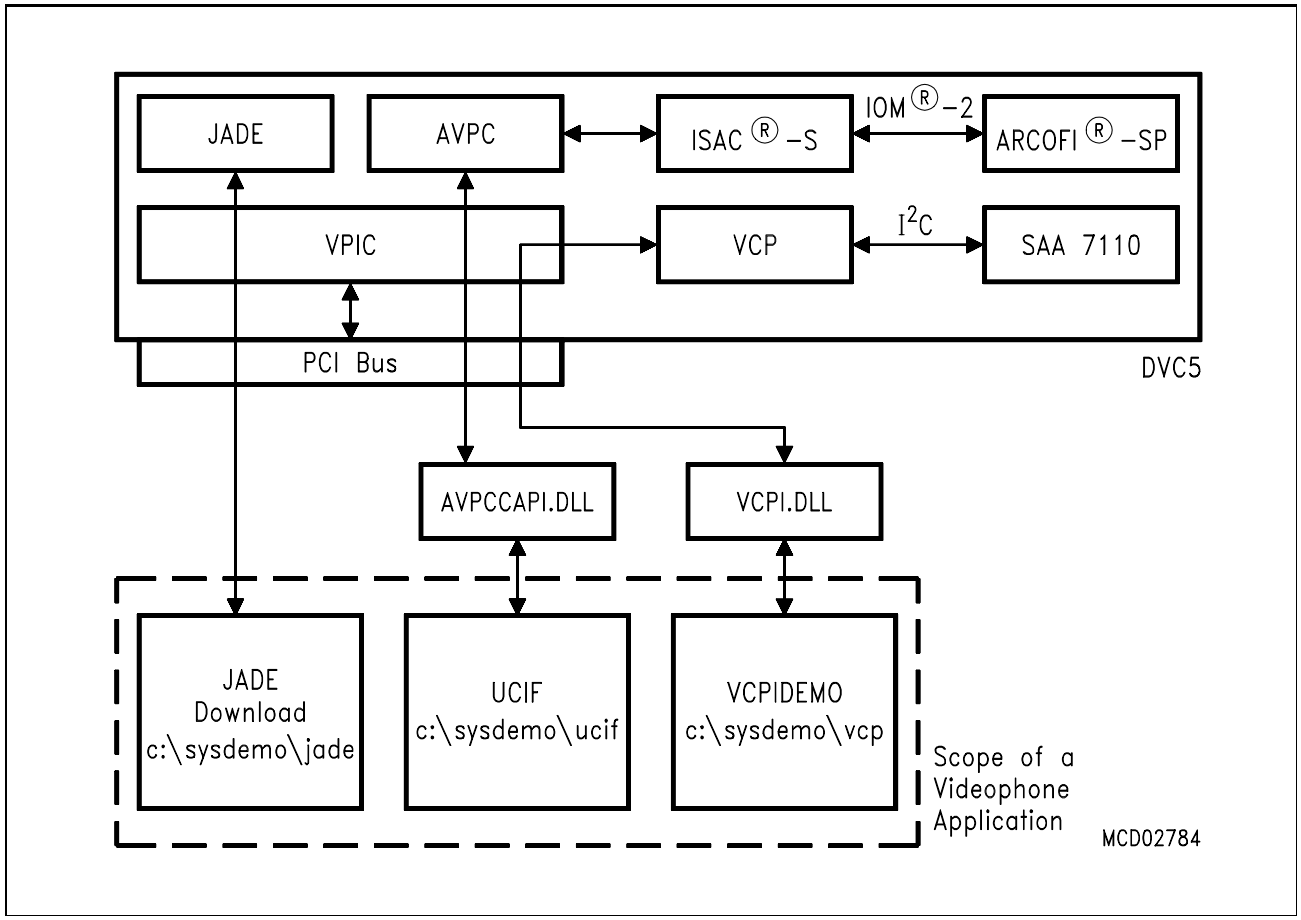

Figure 10
Flow of Software Control for the DVC5-Board

Using AVPC for Videophone Application (DVC5-Board)

The three different flows of software control can be described as follows:

- The program ‘JADE Download’ loads the required firmware into the JADE chip.
- The User Call Control Interface Application (UCIF) is used to control the Advanced Video Phone Controller (AVPC) via a simple handshake mechanism implemented by the Windows 3.x Dynamic Link Library AVPCCAPI.DLL. The AVPC handles the ISAC-S, which programs the ARCOFI-SP using the IOM-2 bus. Thus the UCIF application can access the ISAC-S and the ARCOFI-SP.
- The VCPIDEMO program loads firmware into the Video Communications Processor (VCP) via the Windows 3.x Dynamic Link Library VCPI.DLL. This enables the VCP to do H.261 video compression, multiplexing of video/audio according to H.221 and to protocol control H.242. The SAA 7110 chip is controlled by the VCP via I^2C.

In addition to the ‘Video Reference Board (DVC5-Board) Package SIPB 7280 V2.0’ there is another package named ‘IOS for AVPC Release 2.0 SIPV 6014’ available. Both packages contain the sources of UCIF and AVPCCAPI.DLL. The package ‘IOS for AVPC Release 2.0 SIPV 6014’ contains the source code of IOS for AVPC 2.0 additionally. A functional description of UCIF and AVPCCAPI.DLL is provided in the next two chapters.

3 AVPCCAPI.DLL

3.1 Introduction

This document describes a WINDOWS 3.x Dynamic Link Library (DLL) with name AVPCCAPI.DLL which is nothing but a simple software interface to the Advanced Videophone Controller (AVPC). The name AVPCCAPI was chosen because this interface is similar to the standard CAPI specification.

The main function of AVPCCAPI.DLL is to transfer data in blocks of 10 bytes each to and from AVPC. It relieves a WINDOWS 3.x application program of all the hardware handshake necessary to control AVPC. **Figure 11** shows how AVPCCAPI.DLL is embedded within the general software structure.

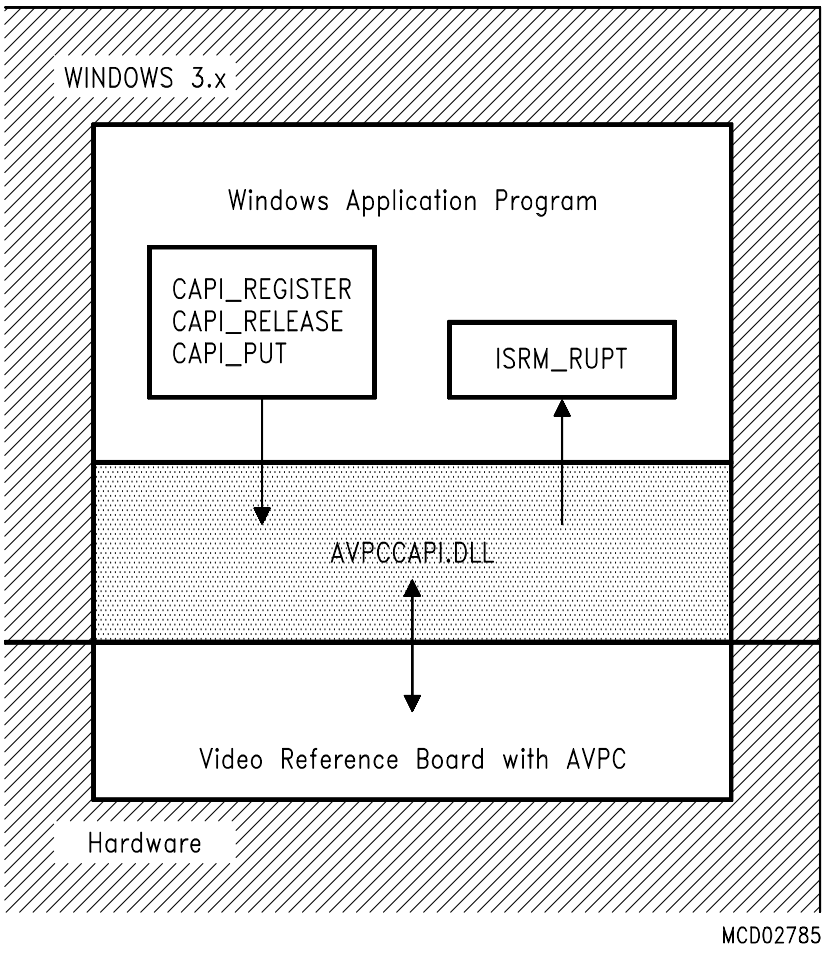

Figure 11
General Software Structure

3.2 Hardware and Software Requirements

To use AVPCCAPI.DLL unchanged, the following hardware environment is required:

1. Standard PC with PCI bus.
2. Video Reference Board containing AVPC
   (Recommended jumper settings of Video Reference Board is specified in Application Note ‘Getting started: Video Reference Board’. This Application Note is delivered with the Video Reference Board.)

To execute AVPCCAPI.DLL, the following software requirements must be met:

3. WINDOWS 3.x0
4. Application program using the functions of AVPCCAPI.DLL e.g. UCIF.EXE (see **Chapter 4**).

3.3 Delivered Files

The following is a summary of files which together implements the AVPCCAPI.DLL.

AVPCCAPI.BAT	Batch file to assemble AVPCCAPI.ASM
AVPCCAPI.MAK	Make file to generate AVPCCAPI.DLL
AVPCCAPI.DEF	Module definition file of AVPCCAPI.DLL
AVPCCAPI.INC	Include file needed from AVPCCAPI.ASM
AVPCCAPI.ASM	File containing assembler source code of AVPCCAPI.DLL
AVPCCAPI.DLL	Dynamic Link Library representing a software interface to AVPC

3.4 Functions and Messages

3.4.1 General Information

To use the functions of AVPCCAPI.DLL in a WINDOWS 3.x application program consider the following hints.

1. AVPCCAPI.DLL has to reside in the current directory, the WINDOWS directory, the WINDOWS/SYSTEM directory or the directories listed in the PATH string of the MS-DOS environment.
2. To access the functions of AVPCCAPI.DLL from a C++ application the functions have to be exported in the following way:

```
extern "C"
{
   int FAR PASCAL _export CAPI_REGISTER(HWND w);
   int FAR PASCAL _export CAPI_RELEASE(void);
   int FAR PASCAL _export CAPI_PUT(LPCSTR MC10DataPtr);
   int FAR PASCAL _export CAPI_GET_IR_COUNT(void);
}
```

   Because of name mangling in the C++ environment the statement extern “C” is necessary.
3. In the module definition file of your application an IMPORT section has to be added to associate the functions above with AVPCCAPI.DLL:

```
IMPORTS AVPCCAPI.CAPI_REGISTER
        AVPCCAPI.CAPI_RELEASE
        AVPCCAPI.CAPI_PUT
        AVPCCAPI.CAPI_GET_IR_COUNT
```

3.4.2 Function CAPI_REGISTER

- Declaration:
  `int FAR PASCAL CAPI_REGISTER (HWND w);`

- Parameter:
  `w` Handle of window whose callback procedure should receive ISRM_RUPT messages. AVPCCAPI.DLL issues this type of message whenever it receives a data byte from AVPC.

- Return Value:
  `0` Error during registration (e.g. AVPCCAPI.DLL is already registered).
  `1` Successful registration.

- Description:
  This function registers AVPCCAPI.DLL i.e. initializing variables, determining direct I/O address of AVPC, setting AVPC handshake signals to a default state and installing an interrupt service routine to receive data from AVPC.

3.4.3 Function CAPI_RELEASE

- Declaration:
```
int FAR PASCAL CAPI_RELEASE (void);
```

- Parameter:
  No parameter required.

- Return Value:
  `0` Error during release (e.g. AVPCCAPI.DLL is already released).
  `1` Successful release.

- Description:
  This function releases AVPCCAPI.DLL i.e. performing of cleanup and restoring the interrupt vector table to the state before CAPI_REGISTER.

3.4.4 Function CAPI_PUT

- Declaration:
```
int FAR PASCAL CAPI_PUT (LPCSTR MC10DataPtr);
```

- Parameter:
  `MC10DataPtr` Pointer to 10 bytes of data which should be sent to AVPC.

- Return Value:
  `0` Successful sending.
  `1` AVPCCAPI.DLL is not registered. Register AVPCCAPI.DLL before sending.
  `2` Timeout during sending.

- Description:
  This function sends a 10 byte data block to AVPC.

3.4.5 Function CAPI_GET_IR_COUNT

- Declaration:
  `int FAR PASCAL CAPI_GET_IR_COUNT (void);`

- Parameter:
  No parameter required.

- Return Value:
  Integer with number of interrupts issued from AVPC.

- Description:
  This function is intended for development purposes only and returns the number of interrupts issued from AVPC.

3.4.6 Message ISRM_RUPT

- Declaration:
  ```
  #include <WINDOWS.H>
  #define ISRM_RUPT WM_USER+255
  ```

- Description:
  The `ISRM_RUPT` message is issued from AVPCCAPI.DLL via the WINDOWS primitive `PostMessage()` whenever it receives a data byte from AVPC. The data byte is passed within the message in the least significant byte of parameter `wParam`. The message is directed to the callback procedure of the window whose handle was announced during `CAPI_REGISTER`.

3.5 How to Generate your own AVPCCAPI.DLL

AVPCCAPI.DLL is a ready to use Dynamic Link Library (DLL) representing a standardized software interface to AVPC. If you want to customize this DLL, make your appropriate changes in source file AVPCCAPI.ASM. Pay attention to the following steps to generate a new AVPCCAPI.DLL.

Create a new directory (e.g. AVPCCAPI).

Copy all the files summarized in chapter ‘Delivered Files’ to this directory.

Set the variable PROJPATH located in makefile AVPCCAPI.MAK to the path mentioned above, e.g.:

```
PROJPATH = C:\AVPCCAPI
```

Use batch file M.BAT to assemble AVPCCAPI.ASM. To meet the requirements of this step a Microsoft Assembler MASM V5.1 or higher is recommended.

Use makefile AVPCCAPI.MAK to generate your own AVPCCAPI.DLL. To meet the requirements of this step a Microsoft Visual C Software Development Package MSVC V1.0 or higher is recommended.

3.6 Technical Details

This chapter gives a detailed description what's behind the functions of AVPCCAPI.DLL. It is not intended for the programmer who only wants to invoke the functions. By way of contrast it is very helpful for the user who wants to make changes in AVPCCAPI.DLL. To achieve best results use this chapter in conjunction with assembler source file AVPCCAPI.ASM.

3.6.1 Register AVPCCAPI.DLL

Figure 12 shows the program flow during registration of AVPCCAPI.DLL. First of all the passed window handle is saved in the variable `hWndApp`. Then the subroutine `GetAddress` determines the base address of the Video Reference Board in the PCs direct I/O space. The Board provides one interrupt line to the PC. The assignment of this interrupt line to the corresponding line of the PCs Programmable Interrupt Controllers (PICs) is done via the PCI BIOS. The number of this line is kept in a register of the VPIC chip contained on the Video Reference Board. VPIC represents the interface of the Board to the PCI bus. The subroutine `GetInterrupt` calculates the absolute interrupt vector composed of the interrupt line number and the PICs vector base. The vector base is determined via a call to the DOS Protected Mode Interface (DPMI). Afterwards the subroutine `InstallHandler` saves the address of the current interrupt service routine associated with the calculated absolute interrupt vector. Then it installs the interrupt service routine named `TheISR` for this vector via DPMI. Next the subroutine `InitSignals` sets the handshake signals of AVPC to a default state. Finally the mask which corresponds to the interrupt line of the Video Reference Board is enabled in the PICs mask register.

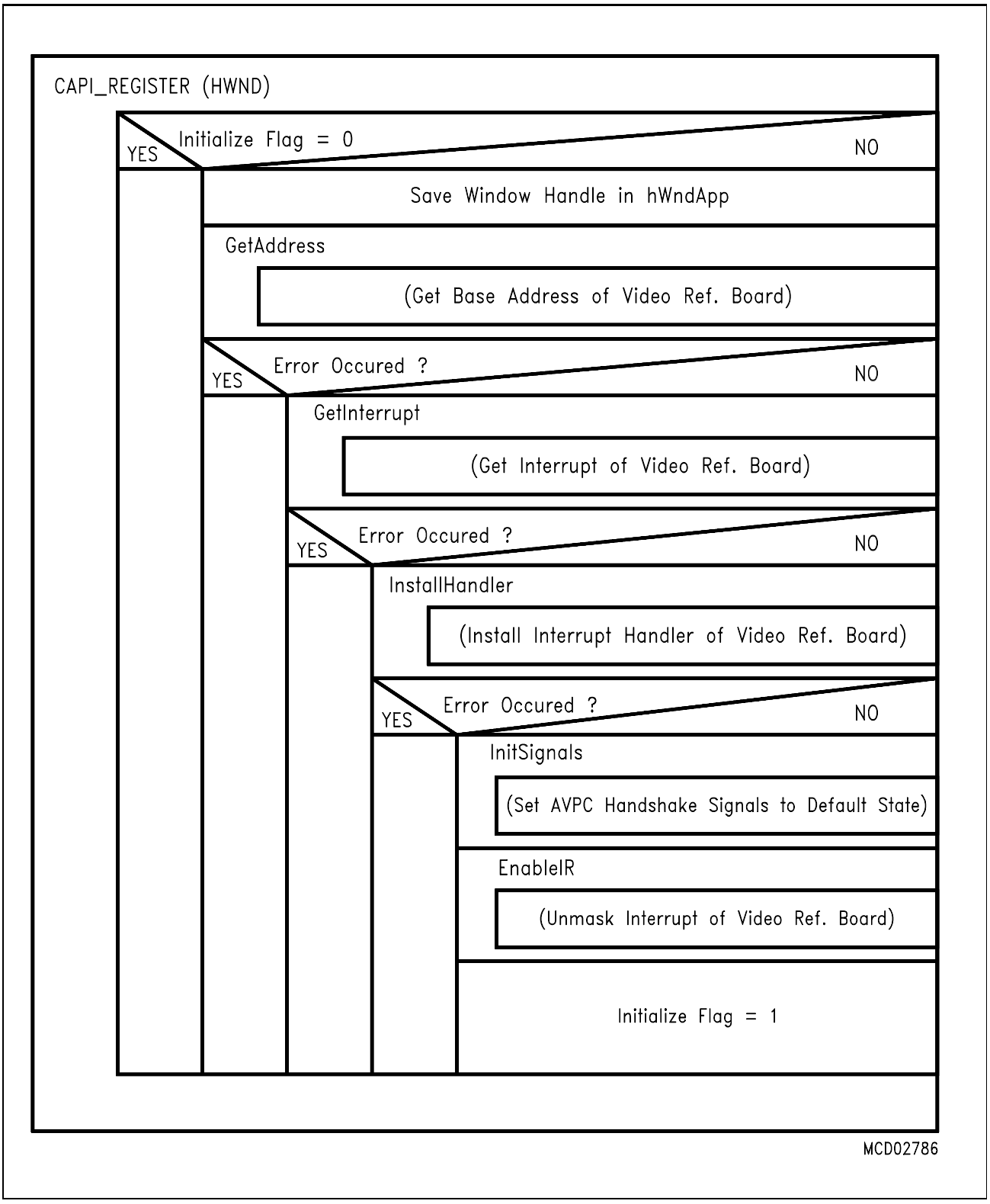

Figure 12
Program Flow of CAPI_REGISTER

3.6.2 Release AVPCCAPI.DLL

Figure 13 shows the program flow during release of AVPCCAPI.DLL. Interrupts issued from the Video Reference Board are controlled from VPIC. Although there is only one interrupt line available multiple interrupt sources like the AVPC interrupt are possible. First of all the subroutine `CAPI_DIS_VPIC` disables the interrupt associated with AVPC in VPIC. Then the mask which corresponds to the interrupt line of the Video Reference Board is disabled in the PICs mask register. Finally the subroutine `DeInstallHandler` installs the previous interrupt service routine via DPMI whose address was saved during `CAPI_REGISTER`.

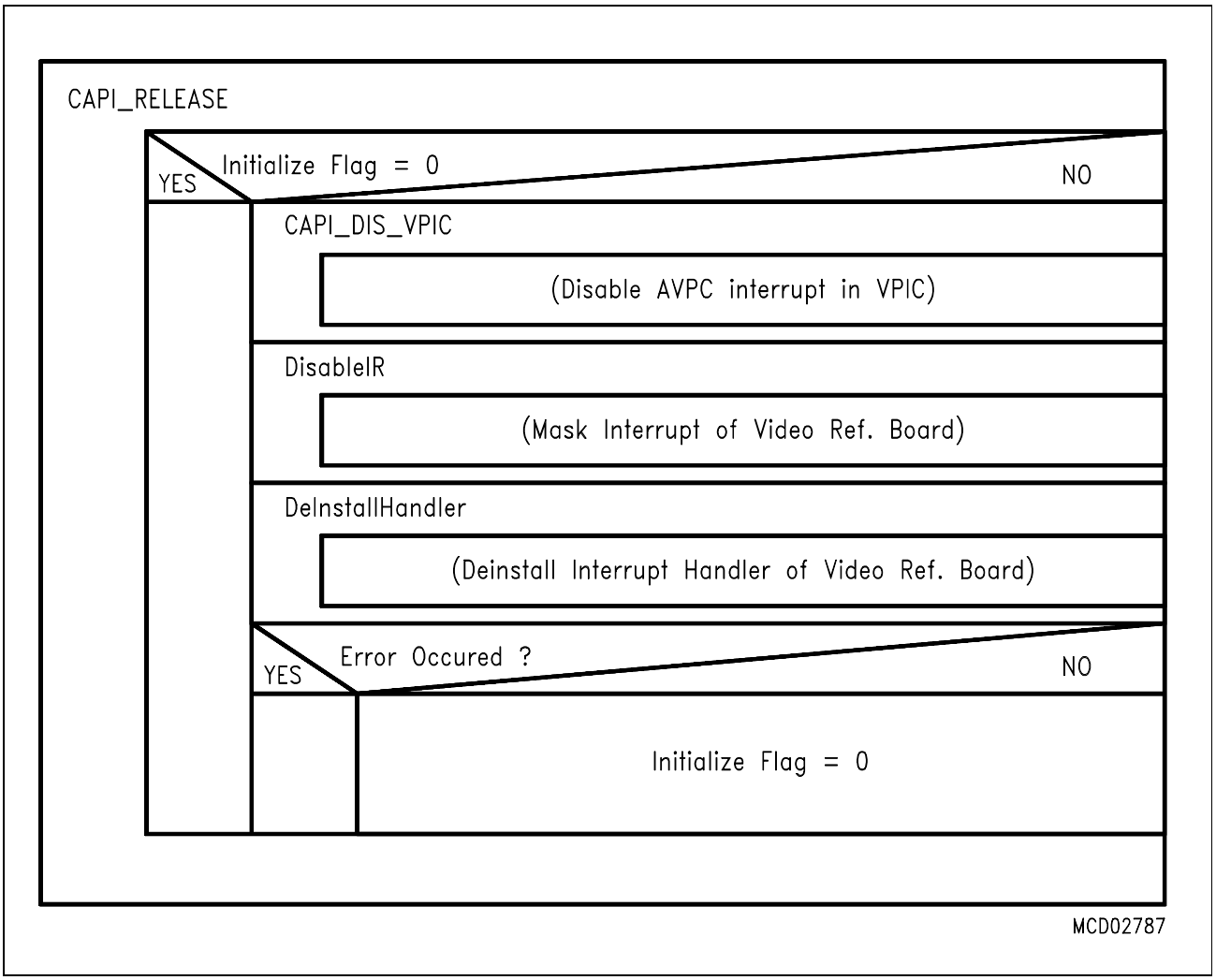

Figure 13
Program Flow of CAPI_RELEASE

3.6.3 Send Data to AVPC

Figure 14 shows the program flow during sending of a 10 byte data block to AVPC. First of all the subroutine `PutFifoPut` copies the data block to an internal FIFO. Then the block is sent to the AVPC in the subroutine `InitiateBlockSend`. Sending is performed bytewise with a special handshake protocol based on a send/acknowledge mechanism. If AVPC does not acknowledge within 4 seconds a timeout is issued.

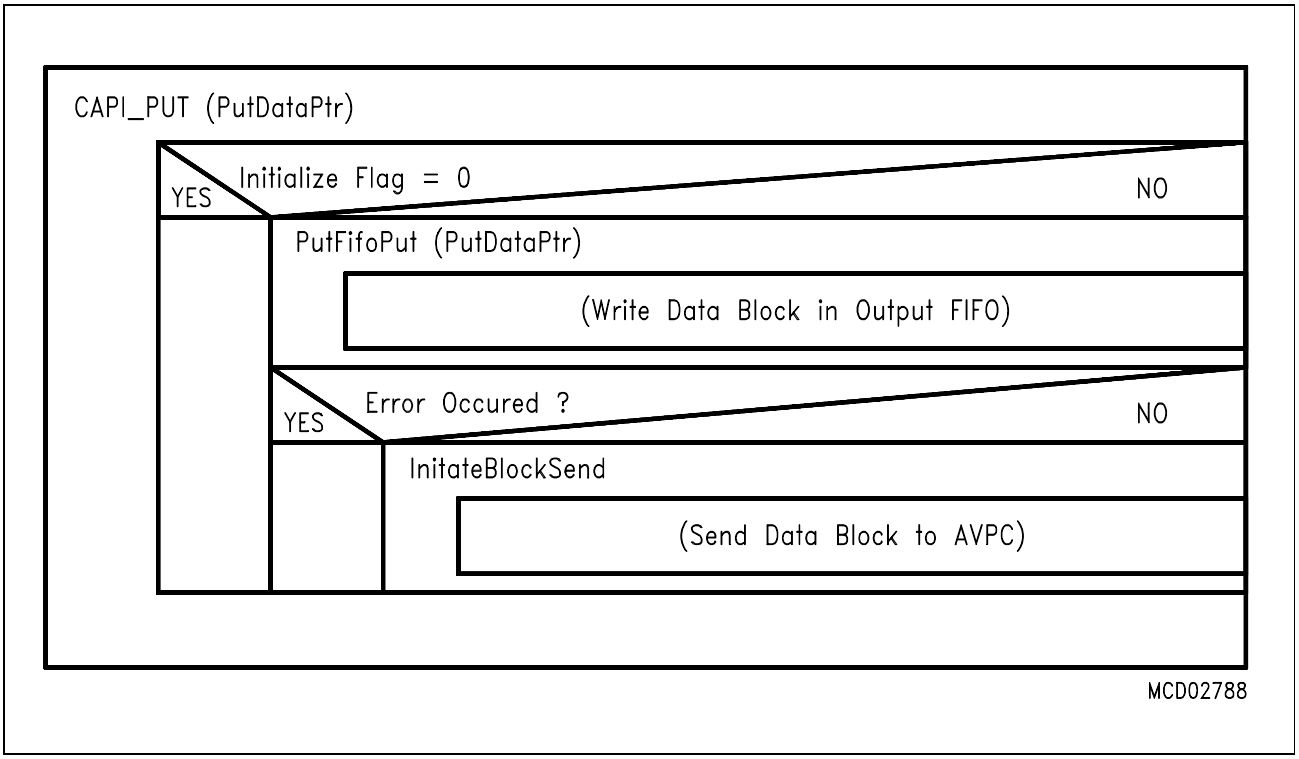

Figure 14
Program Flow of CAPI_PUT

3.6.4 Receive Data from AVPC

Figure 15 shows the program flow when AVPCCAPI.DLL receives an interrupt. If the Video Reference Board issues an interrupt to the PC, the interrupt service routine named `TheISR` is automatically invoked, if previously installed with `CAPI_REGISTER`. First of all the processor registers are saved on stack. Then the interrupt counter `IrCount` is incremented. To identify the interrupt source the interrupt status register of VPIC is read. At present only the AVPC interrupt is handled. If AVPCCAPI.DLL should be extended to service further interrupts the changes must be done at this program location. How to implement such changes is directly documented in the source code file AVPCCAPI.ASM. Disabling the interrupt in VPIC is the first step to service an AVPC interrupt. Next the data byte is received from AVPC. To complete receiving an acknowledge is issued to AVPC. Afterwards a message is sent via the WINDOWS primitive `PostMessage`. The message is directed to the callback procedure of the window whose handle was announced and saved during `CAPI_REGISTER`. The received data byte is passed within this message. In order to release the interrupt an End Of Interrupt (EOI) command is given to the Programmable Interrupt Controllers (PICs) of the PC. Finally the AVPC interrupt is enabled in VPIC, the processor registers are restored and `TheISR` returns from interrupt.

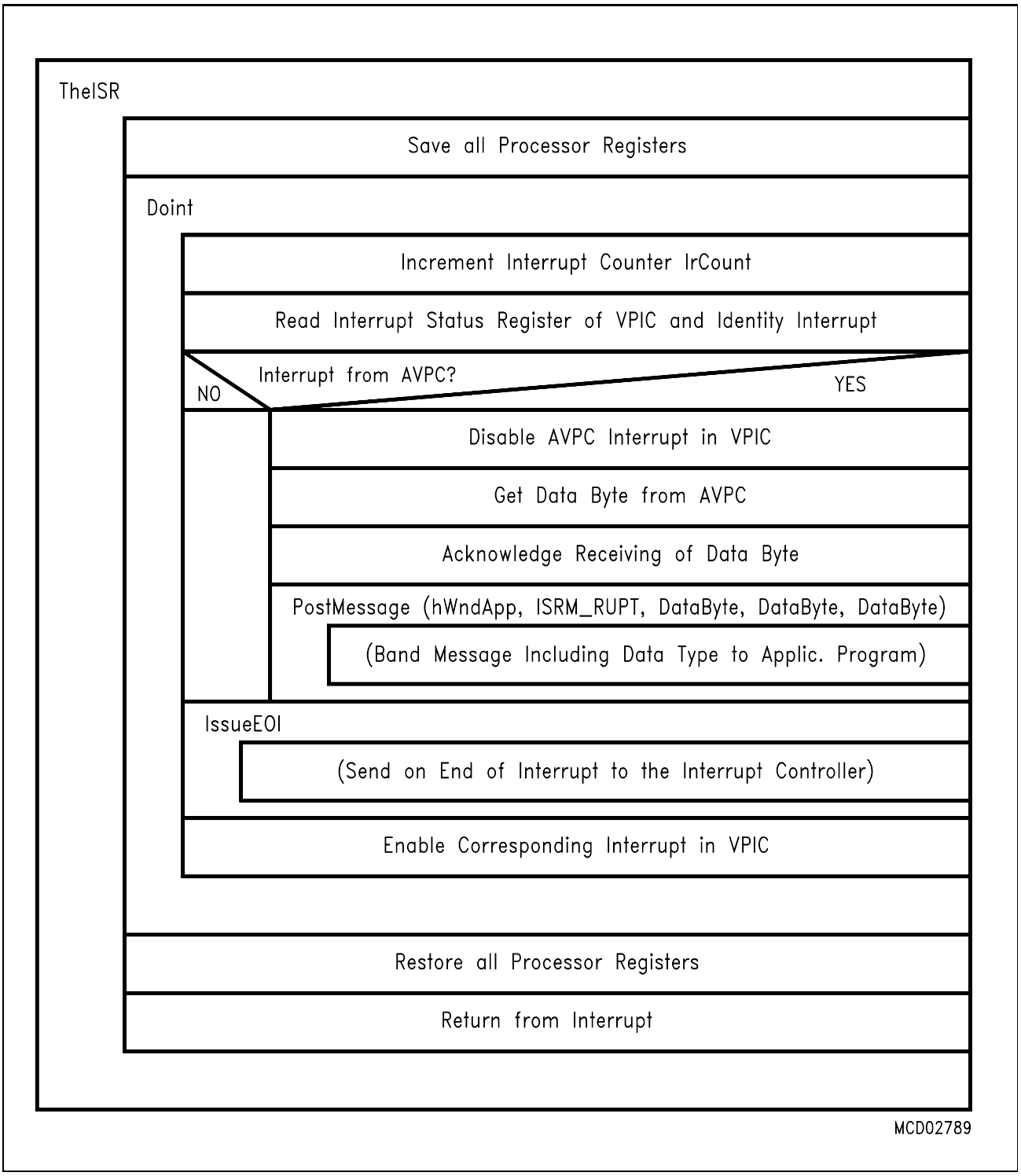

Figure 15
Program Flow of TheISR

3.7 Application Example

UCIF.EXE represents a typical application example which shows how to use the functions of AVPCCAPI.DLL. UCIF provides an interface to the ISDN D channel protocol software executed in the AVPC. The entire functionality of UCIF is described in the next chapter.

4 UCIF

4.1 Overview

This document explains the structure of the C++ source code of the User Call Control Interface (UCIF). This Windows application example is intended for programmers who want to use UCIF as a starter kit for development of their own individual PC Call Control User Interface.

UCIF is an ’orthodox’ Microsoft Visual C++ (V1.0) project based on the Microsoft Foundation Class Library V2.0. It was built on top of the starter application which is automatically created by the AppWizard when the following options are set: Multiple Document Interface (MDI), Initial Toolbar, rest switched off.

Normally, a call control application would not need a document interface. However, here it is useful, as it is intended to install a direct programming path for the Siemens ARCOFI-SP chip. Standard .ARC text files available from ARCOS-SP software can be opened in UCIF’s document frame window, can be edited, sent to ARCOFI-SP and saved on disk again.

There are only some simple add-ons to the basic AppWizard product: Some additional menus in the frame windows, and an additional separate window (‘Call Control’) which is just a modeless dialog box (i.e. cannot be closed by clicking).

Another add-on is the simple class `CCAPI_IO` providing message IO to/from the Advanced Videophone Controller (AVPC) via AVPCCAPI.DLL. Furthermore, a special class CArcofiProg facilitates programming of the ARCOFI-SP. Those are all of the UCIF secrets.

This document only describes the parts of the code which have been added to AppWizard's skeleton starter application. For details on the basic parts of the starter files, please refer to Microsoft's Visual C++ documentation. More detailed descriptions about the SW structure can be found as comments in UCIFs source code files.

Throughout this document, block diagrams are used to visualize the most important parts of the classes implemented in UCIF. **Figure 16** illustrates the drawing rules of those diagrams. The constructor (= class name) is noted in the top field which is framed by bold lines. The right half of this field symbolizes the destructor (~). Function as well as data members of the class are framed by rectangles (data members with dotted lines). When the rectangle touches the rim of the classes' block, the member is public (= accessible from outside). Otherwise it is private or protected.

Rectangles with round edges located outside the class block symbolize Windows message queues / message dispatchers. Circles represent user interface objects like buttons or menus. Public member functions of a class which are pointed to by arrows coming from those circles represent command message handlers.

Note: The user manual of the UCIF.EXE application is described in file UCIFUSR.WRI. This file can be read with the Windows Write Utility.

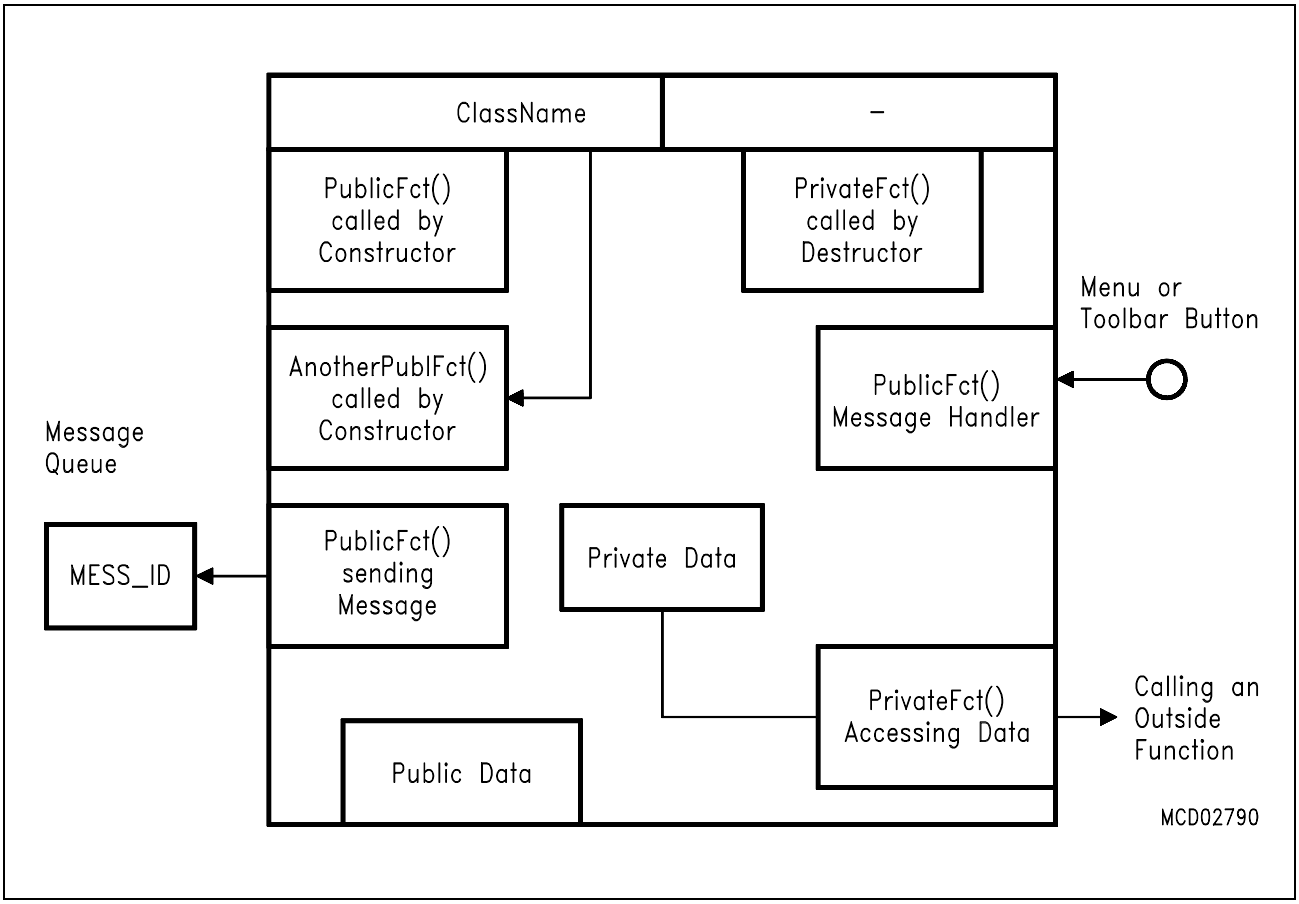

Figure 16
Drawing Scheme for Class Description

4.2 Source Files of UCIF

4.2.1 UCIF.H / .CPP

are the main application source files. UCIF.CPP creates 'theApp' object of class CUcifApp. TheApp holds public pointers to the main objects of UCIF: The Main Window, the Call Control, the CAPI_IO and the Test Dialog object.

The override version of CUcifApp's InitInstance() member creates the main objects on the heap ('new') and stores the pointers. Additionally in InitInstance()and ExitInstance()a HW Reset is made to ensure an initialized state of AVPC.

4.2.2 MAINFRM.H / .CPP

contain the frame class CMainFrame, which is derived from CMDIFrameWnd and controls all MDI frame features, like menus and toolbars. All handlers for ARCOFI-SP programming are also included in CMainFrame. They are implemented as standard Windows command handlers. Furthermore, CMainFrame contains the parser functions for ARCOFI command files (extension .ARC). The MDI surface of CMainFrame allows you to open, edit and save .ARC files. Open files can be sent to the ARCOFI-SP through the parser functions (menu command).

In addition, the MAINFRM files define the special CArcofiProg class, which supports register programming of the ARCOFI-SP chip.

4.2.3 UCIFDOC.H / .CPP
UCIFVIEW.H / .CPP

contain the CUcifDoc and CUcifView classes. They contain nearly 100% original AppWizard code. The only differences are:

CUcifView is derived from CEditView instead of CView, introducing standard text editing features (incl. clipboard) for .ARC file access. CUcifDoc's Serialize() member function was overridden in order to make use of SerializeRaw() operation of CEditView. Thus, UCIF's document class is nothing more than a (mostly empty) 'transit station' for the file data which CUcifView possesses by itself.

4.2.4 CALLDLG.H / .CPP

define the main user surface part of this application: The Call Control object and window. The `CCallDlg` class is derived from `CDialog`. When created at program startup (in `theApp`'s `InitInstance()`), it presents a modeless (persistent) dialog box window to the user offering two independent call control fields. The window was designed using Visual C++'s AppStudio tool (file UCIF.RC contains the ressources).

The `CCallDlg` object embeds two objects of class `CStateMach`, which is also defined in the CALLDLG files. The state machine is needed to inform the user of the current call state, and to automatically choose appropriate programming sequences for the ARCOFI-SP (like dial tone generation, B channel switching etc.).

4.2.5 TSTDLG.H / .CPP

define the classes `CCAPI_IO` and `CTstDlg`. The class `CCAPI_IO` is the interface to an external driver module in DLL format (here: AVPCCAPI.DLL). Upon construction / destruction, a `CCAPI_IO` object automatically sends register / release calls to the DLL.

The `CTstDlg` class (derived from `CDialog`) is nothing more than a simple monitor for the data primitives sent to/from the DLL. In test mode without DLL linked (i.e. with NO_DLL compiler switch set), it's 'get' edit box serves as a user input field for simulated indications. The `CCAPI_IO` members call `CTstDlg` functions when necessary. Furthermore, there is a possibility to write all messages between UCIF and AVPC in a fixed file (see menu: view->monitor).

4.3 Compiler Switches

The following compiler switches are available in the UCIF source files:

NO_DLL — When defined, it is assumed that no AVPCCAPI.DLL is present. This helps testing UCIF functions without AVPC. The `Do_capi_...()` functions of `CCAPI_IO` do no longer call AVPCCAPI.DLL functions. To simulate indications from the DLL, `CTstDlg`'s `capi_get()` is called. With NO_DLL not set, `capi_view_get()` is called instead, just echoing indications, offering no chance to edit them.
When NO_DLL is defined, POST_IR (see below) must **not** be defined!

POST_IR — When defined, indications coming from the AVPC are received bytewise. The DLL sends each byte as a parameter of a separate `PostMessage`. In this case the `CCAPI_IO` object includes neither `GetPrimitive()` nor `Do_capi_get()` functions.
When POST_IR is not defined, AVPCCAPI.DLL is assumed to send an 'empty' `PostMessage` as soon as one complete indication primitive (10 bytes) is received inside the DLL. `CCallDlg`'s `OnCAPIMessage()` handler responds by calling `CCAPI_IO`'s `GetPrimitive()` member. The current version of AVPCCAPI.DLL can only be used with POST_IR set.

COUNTER When defined, counters for request (=put) / indication (=get) primitives are visible in the status bar of the frame window. Upon each request, `CCAPI_IO's Do_capi_put()` sends an `INCR_PUT_COUNT` message to the `CMainFrame` object. To count indications, `CMainFrame OnCAPIMessage()` handler posts `INCR_GET_COUNT` messages.

Note: This is commented out in the current version, to avoid long response times on 'slow' PCs.

4.4 Classes of UCIF

4.4.1 CMainFrame

A lot of functionality has been added to the `CMainFrame` class which AppWizard derived from `CMDIFrameWnd`. `CMainFrame` is the 'biggest' class in this application. It contains all the handler functions for the application specific user commands like ARCOFI-SP programming, DLL registering and so on. On the other hand, the bundle of parser functions needed to translate .ARC files represent a big part of `CMainframe`'s code. See the private function block '`parseArcofi`...' in **Figure 17** which shows `CMainFrame`'s overall structure.

When `CUcifApp`'s `InitInstance()` constructs the (one and only) `CMainFrame` object - pointed to by `m_pRealFrame` - on the heap, the constructor reads in various fixed .ARC files for initialization and tone programming by calling the private `readArcofiCmdFiles()` function. For fast access later on, the files are put into `CMemFile` objects. In **Figure 17** the message handlers for all ARCOFI-SP menu commands are listed on the right side. The tone handlers (upper section) invoke `SendToArcofi()` with the appropriate `CMemFile` as argument. `SendToArcofi()` calls the parser function `parseArcofiLine()` to generate binary `CByteArray` primitives from the text lines. Before sending the primitive to the DLL via CCAPI_IO's `SendPrimitive()`, it is passed to the `ModCmd()` member of the embedded `CArcofiProg` object in order to locally store ARCOFI's register values.

The two toggle items in the ARCOFI menu, AutomaticMode and Change Law (a/u) just switch boolean flags. Those flags are checked when `CMainFrame`'s `OnSendToArcofi()` handler is called by the state machine in the Call Control window (message id = `AUTO_ARC`) to automatically program ARCOFI-SP dependent on the call state. The only task of `OnSendToArcofi()` is to call in turn the appropriate `OnArcofi...()` handler via `SendMessage(WM_COMMAND, ..)`, simulating a menu click by the user. In contrast to the tone switching commands described above, the IOM-2 switching commands (like B1, B2) do not rely on .ARC files. They just modify single bits in ARCOFI-SP registers using the `SetMode()` and `SetBits()` members of the `CArcifiProg` object. For loop switching commands in ISAC-S menu, fixed primitives are stored in `CByteArrays m_baISAC...`

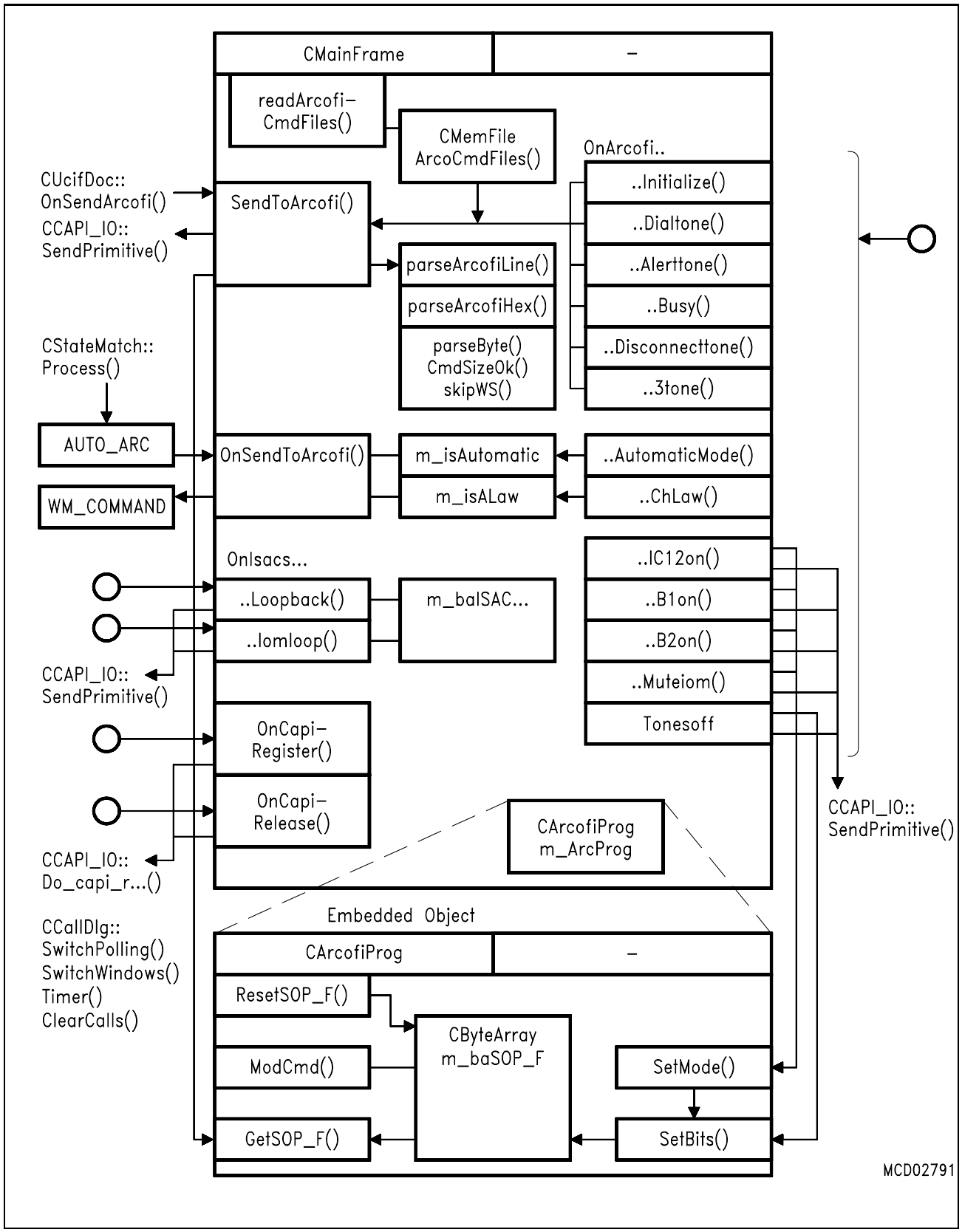

Figure 17
CMainFrame Class

4.4.2 CCallDlg

The CCallDlg object is created as a modeless (persistent) dialog box by CUcifApp's InitInstance(). The class CUcifApp stores a pointer to this object in m_pCallDlg. **Figure 18** illustrates the structure of the CCallDlg class.

CCallDlg provides buttons and corresponding message handlers for the standard call control requests Connect, Clear and Dial. The handlers simply put together request primitives in a private array aRequest[]. Then they call the SendMsg() function, which in turn calls CCAPI_IO's SendPrimitive(). For dialing numbers, the common OnDial() function calls SendMsg() in a loop to put all dial digits in separate primitives.

Indications from the AVPCCAPI.DLL are processed by the OnCAPIMessage() handler. There are two different conventions for this process, depending on the POST_IR compiler switch (see above). Whenever one indication primitive is complete, OnCAPIMessage() calls the private ProcessInd(), which checks if the primitive is an error message from the IOS. In that case, it shows a message box explaining the error.

Both SendMsg() and ProcessInd() pass further processing to one of the two embedded CStateMach objects by calling either DecodeRq() or DecodeInd(). The state info strings returned by the Decode...() functions are used to update the Call State field in the Call Control window.

The private Process() member of CStateMach (**Figure 19**) keeps track of the call state. It does two things: It creates and returns clear text strings for user information, and it posts messages to the OnSendToArcofi() handler of the main frame in order to have the ARCOFI-SP chip programmed according to changes of the call state (e.g. channel switching, tone generation etc.). When CCallDlg's constructor creates the two CStateMach objects, it registers the main frame pointer as target (m_pTarget) for the PostMessage() calls used for ARCOFI-SP programming.

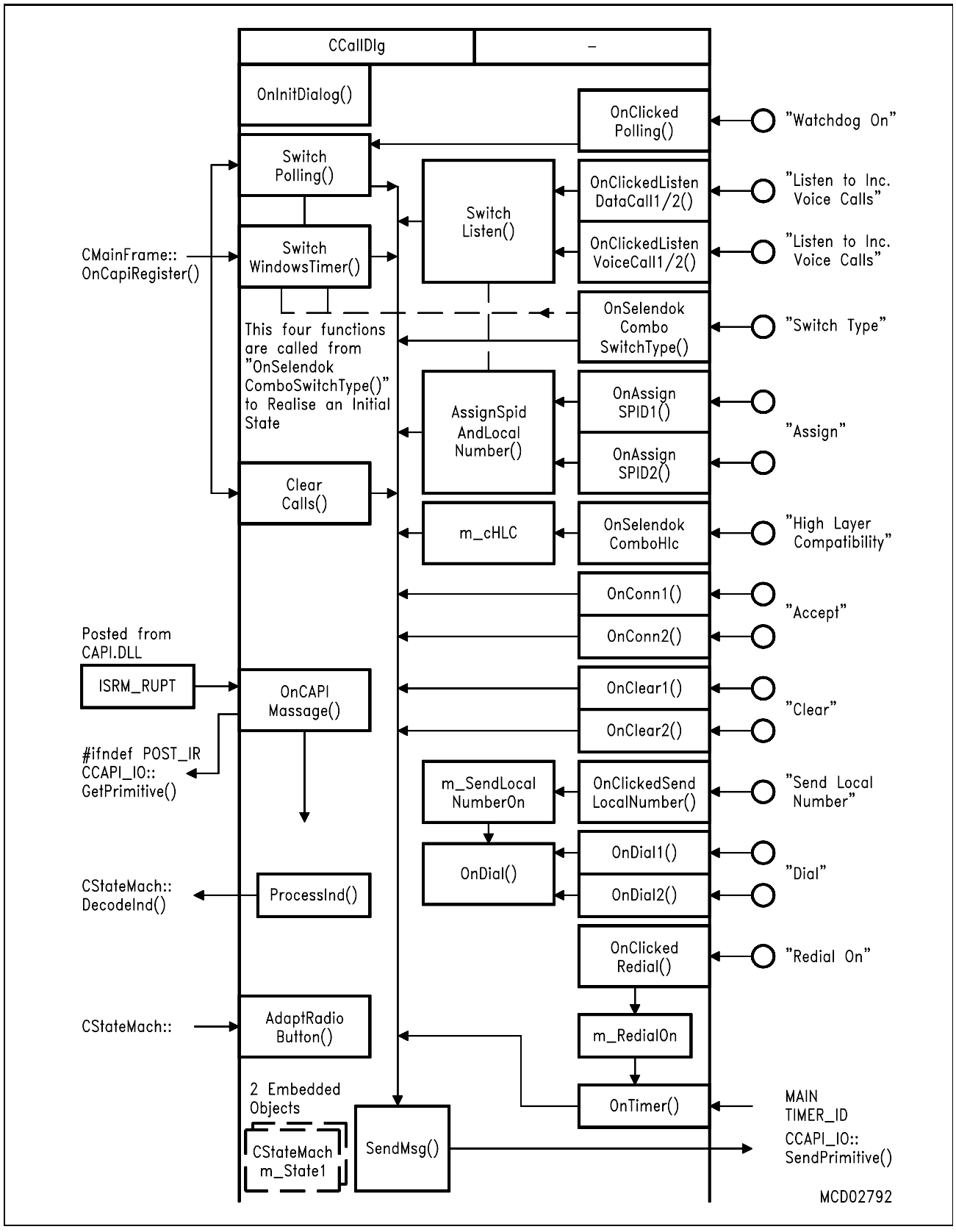

Figure 18
The Call Control: CCallDlg Class

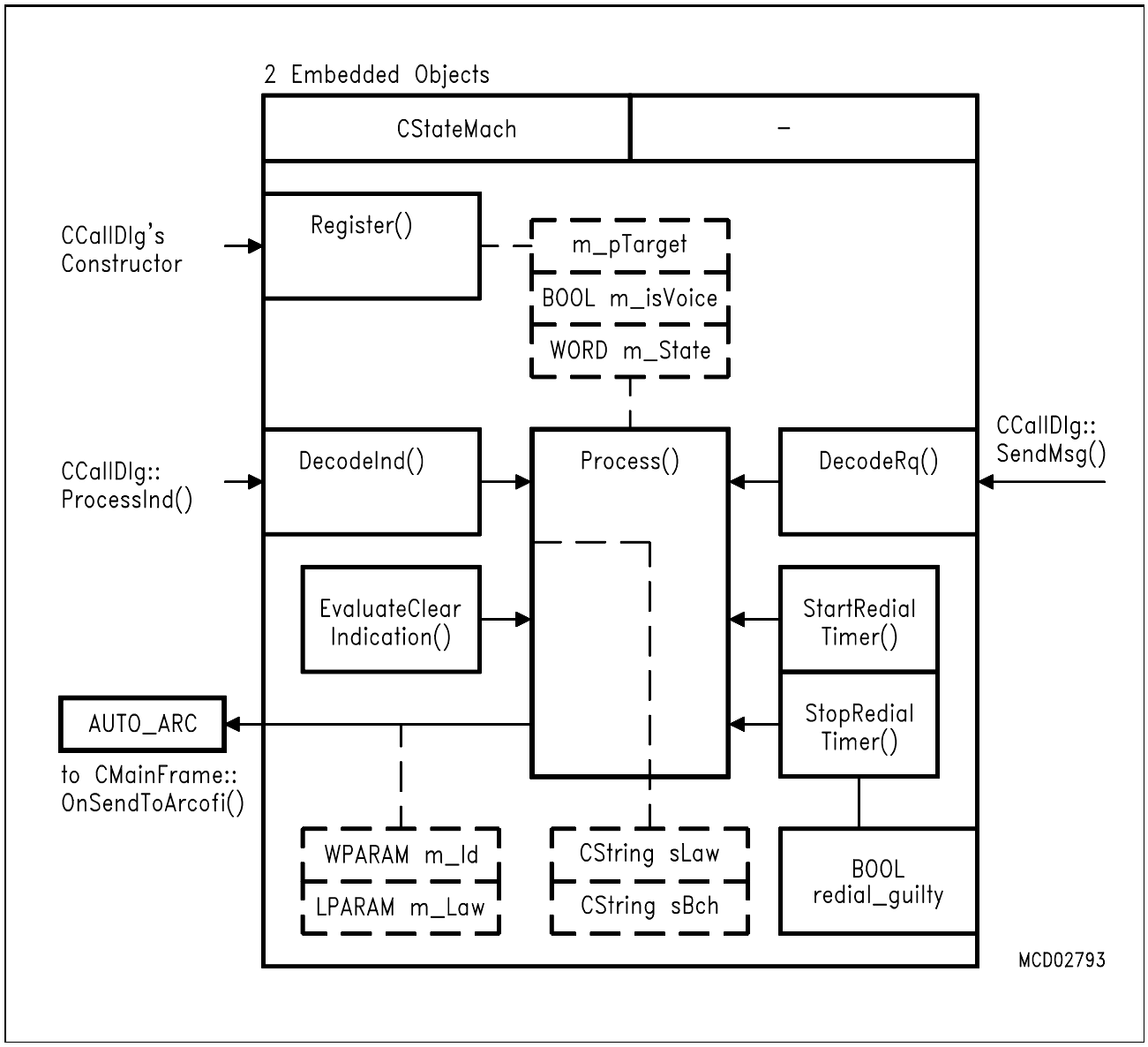

Figure 19
The State Machine: CStateMach

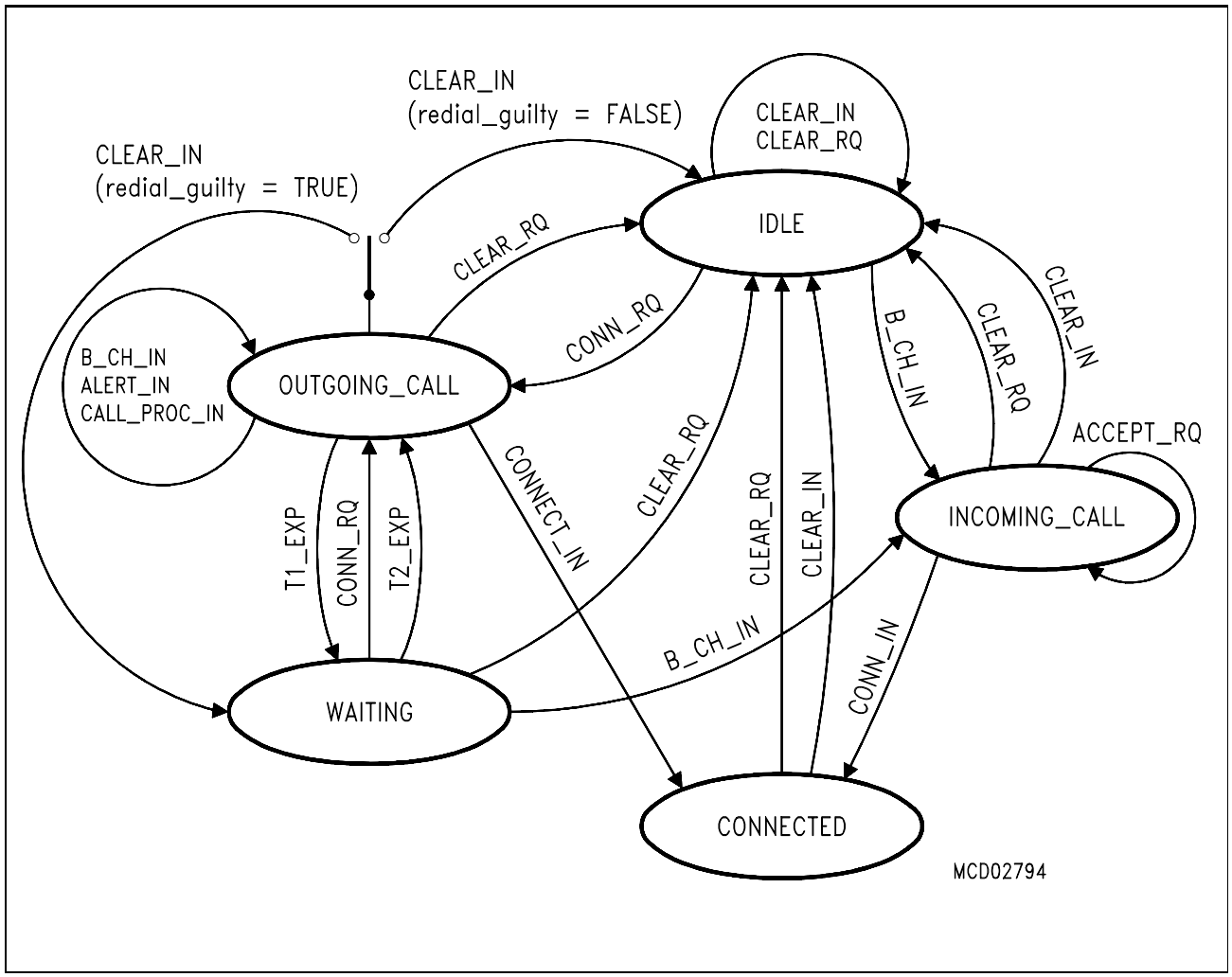

Figure 20
Flow Diagram of CStateMach

4.4.3 CCAPI_IO

The CCAPI_IO class (as illustrated in **Figure 21**) encapsulates all the interface functions needed to send request primitives to AVPC via DLL, and to receive indications from the DLL. When a CCAPI_IO object is created, it tries to register itself by calling the DLL's capi_register() function. With this call, a Windows handle (HWND) to the object which should be the target for messages posted from DLL is passed. HWND must be given to CCAPI_IO upon its own creation. In this UCIF application, the HWND of the CCallDlg object is used. Thus, CCallDlg's OnCAPIMessage() handler is the target for DLL's PostMessage(ISRM_RUPT).

CCAPI_IO keeps a local private copy of the currently processed primitive in aMsg[]. When the application on top wants to send a request, it calls SendPrimitive(). Then this function calls DLL's capi_put() via its own Do_capi_put(), while checking if the DLL accepts the put operation. If not, it displays an error message box.

For the receiving of indications from DLL, two different ways are possible (see definition of POST_IR switch above). CCAPI_IO only deals with indications if POST_IR is **not** set. In that case, the application module on top (here: CCallDlg object) calls GetPrimitive() after receiving a PostMessage() from DLL. CCAPI_IO then calls DLL's capi_get() through Do_capi_get().

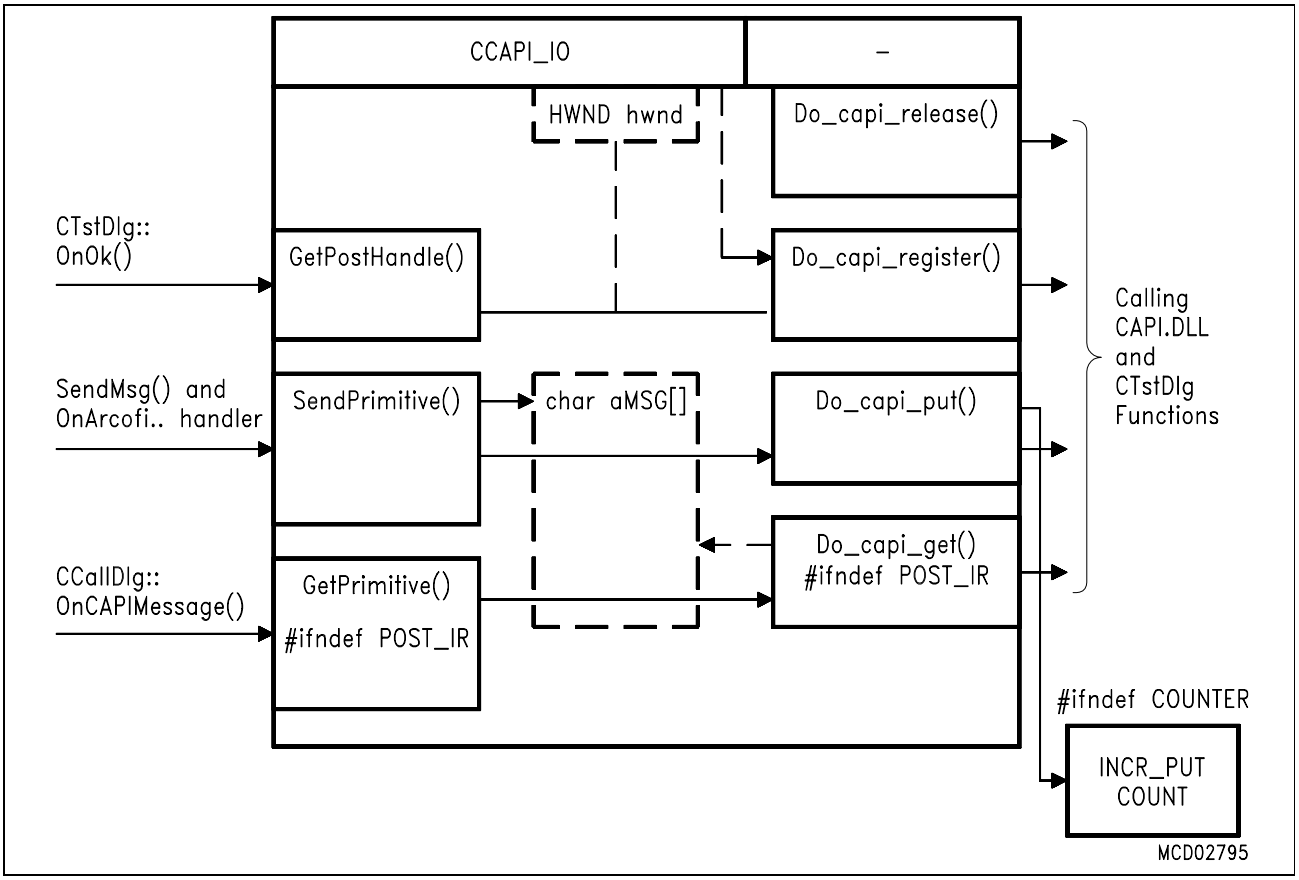

Figure 21
The I/O Interface, CCAPI_IO Class

4.4.4 CTstDlg

`CTstDlg` is a modeless dialog box (like `CCallDlg` is). It is only visible if switched on in the View menu of the main frame window. The `capi_...()` functions of `CTstDlg()` are called by `CCAPI_IO`'s `Do_capi_...()` functions depending on the NO_DLL compiler switch (see above). **Figure 22** illustrates the structure.

When NO_DLL is set, i.e. UCIF is operated without DLL, `CTstDlg()` serves as a source for simulated indications which can be entered in an edit box ('get') in hex format. Then the override version of the dialog box's `OnOK()` handler issues a `PostMessage(ISRM_RUPT)` identical to the one the DLL would issue if it is present.

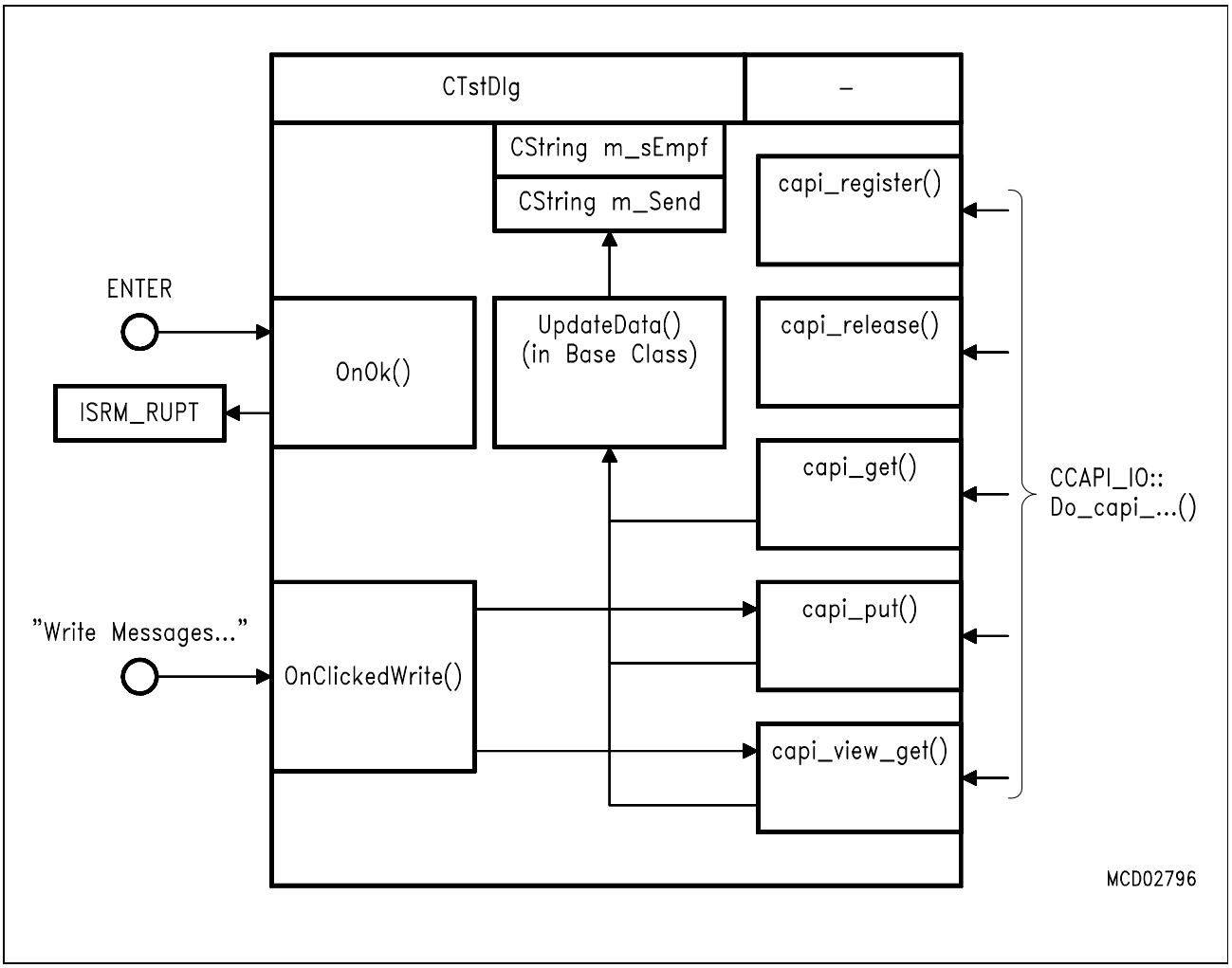

Figure 22
The DLL Monitor, CTstDlg Class