RZ 3446 (# 93753) 08/19/02
Electrical Engineering 98 pages

Research Report

Requirements of a Generic Segmentation and Reassembly Building Block

Florian Gerbl and Maria Gabrani*

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

*Contact author: mga@zurich.ibm.com

Requirements of a Generic Segmentation and Reassembly Building Block

Florian Gerbl and Maria Gabrani*

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

*Contact author: mga@zurich.ibm.com

Abstract

The main focus of this report is the investigation and analysis of the requirements of a Segmentation And Reassembly (SAR) building block for a Network Processor. The SAR block should be generic enough to accommodate both ATM and IP traffic and cover a variety of protocols and applications. A data unit travelling a network can be split into parts (segmented), combined with other data units (multiplexed), and rejoined into one (reassembled) several times and in different ways during its time in the network. Such a packet manipulation depends on the size of the packet, the Quality of Service (QoS) requirements of the application that transmits the data unit, the type of the network protocol, the transport layer, the rate and configuration of the lines it traverses, and the capabilities of the network nodes, among others. A network node that supports both ATM and IP transport protocols, a number of line configurations and a number of application areas should therefore be able to first recognize and second manipulate (segment, mix and reassemble) data units in a specific way that depends on various network parameters. A SAR function that can be generic enough to handle all these protocols in an efficient way is a considerable advantage, if not a requirement, in a flexible system. However, a prerequisite for its architectural definition is a well defined set of requirements, which is established in this report.

Requirements of a Generic Segmentation and Reassembly Building

Florian Gerbl, Maria Gabrani

Contents

1 Introduction

1.1 Project Description

The main focus of this internship is the investigation and analysis of the requirements of a Segmentation And Reassembly (SAR) building block for a Network Processor. The SAR block should be generic enough to accommodate both ATM and IP traffic and cover a variety of protocols and applications.

1.2 Motivation

A data unit traveling a network can be split into parts (segmented), combined with other data units (multiplexed), and rejoined into one (reassembled) several times and in different ways during its time in the network. Such a packet manipulation depends on the size of the packet, the Quality of Service (QoS) requirements of the application that transmits the data unit, the type of the network protocol, the transport layer, the rate and configuration of the lines it traverses, and the capabilities of the network nodes, among others.

For example, if IP packets are to be sent via an ATM network, the packets must be transformed into cells. This can be accomplished using the appropriate ATM Adaptation Layer (AAL), depending on the packet size and the QoS requirements. The size of cells is 53 bytes, made up by a 5 byte header and 48 bytes of payload. The size of the packets can vary from a few bytes up to several kilo bytes. The QoS may require for example a Constant Bit Rate (CBR) or a real-time Variable Bit Rate (rt-VBR). After the appropriate AAL has been chosen, the IP packet is segmented or multiplexed accordingly, in a way that ensures its safe transfer in the network and its correct reception in the end host.

Another example is the case of wireless networks. In an IP wireless network a voice packet is approximately 30 bytes long. Such a packet size is very small to be handled independently by certain network nodes (Node Bs). Therefore, the voice packets that are going to the same direction are usually (PPP) multiplexed with other packets. The line rates in the "first mile" of the wireless network are not very large ($\sim$10x Mbps). For economy reasons the wireless network providers use E1/T1 lines (2 Mbps) that they bundle together to create higher rates, e.g. 16 E1/T1 = 32 Mbps. In order to send packets in a E1/T1 bundle the (multiplexed) packets need to be split into smaller segments (PPP MP). Of course the multiplexing and the segmentation need to be done in such a way so that the initial voice packets can be fully recovered on the other side of the network.

A network node that supports both ATM and IP transport protocols, a number of line configurations and a number of application areas should therefore be able to first recognize and second manipulate (segment, mix and reassemble) data units in the required way. The required way is highly dependent on several parameters, as described in the previous paragraphs. Therefore, a SAR function that can be generic enough to handle all these protocols in an efficient way is a considerable advantage, if not a requirement, in a flexible system.

Before its architectural definition a well defined set of requirements is necessary. This important task is the assignment of this student internship.

2 Overview of examined protocols

2.1 PPP Multiplexing (PPP MUX)

The method of PPP Multiplexing offers the possibility of sending multiple PPP encapsulated packets in a single PPP frame, thus reducing the PPP overhead per packet. A tradeoff has to be made between overhead (which decreases with increasing number of subframes per frame) and delay being introduced at the transmitter (which increases with increasing number of subframes per frame).

According to [5] a PPP MUX frame consists of a PPP header, one or more *subframes* and a *Frame Checksum (FCS)*.

Each subframe consists of a *Length Field*, a *Protocol Field* and an *Information Field* which carries the information that shall be transmitted from the PPP MUX transmitter to the receiver.

The Length Field comprises the *Protocol Field Flag (PFF)*, the *Length Extension (LXT)* and the *Subframe Length (LEN)*.

The Protocol Field carries the *Protocol Identifier (PID)* corresponding to the following Information Field. In order to reduce overhead the Protocol Field can be left out if two or more subsequent subframes otherwise had the same PID value. Only the first one of those subsequent subframes contains the PID, within the others it is not present. If the Protocol Field is omitted, PFF=0 indicates its absence, otherwise PFF equals 1. If the Protocol Field is included, it should be compressed (i.e. the upper byte is simply omitted if it equals zero). This compression is independent of the negotiation of PFC (Protocol Field Compression), which only affects the width of the PID in the PPP MUX header.

The width of LEN is either 6 bit or 14 bit, depending on the value of LEN. If LEN is 6 bit wide, LXT=0, otherwise it equals 1.

The flow graph in Figure 1 illustrates the way a PPP MUX transmitter works. Figure 2 shows the receiver. Please consult [5] for more details.

2.2 PPP Multilink Protocol (PPP MP)

The PPP Multilink Protocol provides the functionality of splitting a data packet into fragments, sending the fragments over different links, receiving and reassembling them, thus increasing the bandwidth. The different links are allowed to have different rates and there is no special requirement for the fragments' sizes, neither for their absolute nor their relative sizes. The probability that fragments are getting reordered is extremely small. To achieve its goals, PPP MP makes use of a 4(2) byte sequencing header.

According to [7] a PPP MP frame consists of a *PPP header*, an *MP header*, the *Fragment Data* and the *PPP Frame Checksum*. The Fragment Data contains one part of the encapsulated original data packet. This encapsulation, which is done

before the splitting, is described in [7] as a normal PPP encapsulation with Address and Control Field Compression (ACFC) and Protocol Field Compression (PFC) switched on. The RFC also states that ACFC and PFC can be negotiated independently on each link. That might cause some confusion. The following sentences clarify what is meant there. The transmitter gets a packet that shall be sent using PPP MP. It takes the respective PID, compresses it if possible (i.e. omits the first byte if it equals zero), and prepends it to the packet. This is what is meant by encapsulation with ACFC and PFC in effect. No Frame Check Sum (FCS) is added, since the FCS is part of the framing and not part of the encapsulation. The resulting data unit (PID plus data packet) is split and the fragments are then sent over the multiple links after having been encapsulated according to the options (ACFC, PFC, ...) that have been negotiated on each link individually.

The MP header consists of a B bit, an E bit, 6 (default) or 2 filler zeros and a *Sequence Number*. The Sequence Number can be negotiated to be either 14 bit (default) or 6 bit wide. Depending on this width the whole MP header is 2 or 4 Bytes long.

B=1 indicates that the fragment delivered in the current MP frame contains the first part of the original packet. E=1 indicates that it is the last part. Every combination of B and E values is valid in a single MP frame.

2.3 The Multi-Class Extension to Multi-Link PPP (PPP MC)

In a way PPP MP supports two levels of priority. If there is high priority data to be sent it can be put into normal PPP frames (instead of PPP MP frames) and sent between the PPP MP frames. Thus high priority data can suspend low priority data. For many applications two levels of priority are not sufficient. Furthermore the high priority data can only be sent as it is and cannot be fragmented to be transmitted over multiple links. The Multi-Class Extension to Multi-Link PPP as described in [1] solves this problem. It offers the user 16(4) levels of priority by using 4(2) of the MP header's filler zero bits to define classes. Using PPP MC is nothing else than running an own instance of PPP MP for every single class with some additional control functions.

Since PPP MP can be interpreted as running PPP MC with class number zero and without the Prefix Elision option that is applicable when running PPP MC, we will not explicitly investigate PPP MP throughout the rest of this document.

The flow graph in Figure 3 illustrates the way a PPP MC transmitter works. Figure 4 shows the receiver. Please consult [1] for more details.

2.4 ATM Adaptation Layer Type 1 (AAL 1)

AAL 1 provides the means necessary for transmitting data that requires a constant bit rate, e.g. uncompressed audio and video. AAL 1 is subdivided into two sublayers: the Convergence Sublayer (CS) and the Segmentation and Reassembly (SAR) Sublayer. The functions of the CS depend very much on the application AAL 1 is

being used for. Timing is one of its most important tasks. For this reason the CS is not considered to be part of the SAR building block. The remaining sublayer, AAL 1 SAR, adds a one byte header to a data unit on the transmit side and removes it on the receive side. The 1 byte header consists of the *Sequence Number (SN)* field and the *Sequence Number Protection (SNP)* field. However, the AAL 1 SAR layer does not perform any segmentation and reassembly. Therefore, were consider that the functional requirements of the AAL 1 protocol are covered by the requirements of the other protocols and we omit its description in most parts of this document. However, we infer any special requirements as necessary.

2.5 ATM Adaptation Layer Type 2 (AAL 2)

Since the size of the payload of an ATM cell is fixed to 48 bytes, data being smaller or bigger than 48 bytes has to be adapted to fit into one or more cells. In case of big data it has to be split and recombined. Since splitting and recombining generally requires the use of some kind of a header, the bandwidth-efficiency can decrease. In case of data smaller than 48 bytes another problem occurs. Sending cells without having used all the space available for payload, bandwidth is being wasted. To compensate this, one could send more than one data unit in a single ATM cell. But waiting for a cell to be filled completely introduces delay. AAL 2 is designed to accommodate variable bit rate traffic with timing requirements between source and destination (e.g. mobile traffic). AAL 2 provides the means for multiplexing data streams of more than one AAL 2 user, which means that a single ATM cell can convey data belonging to more than one user. Since this reduces the time it takes to fill a cell, even delay sensitive applications can use AAL 2 without wasting bandwidth.

AAL 2 consists of two major sublayers: the Segmentation and Reassembly Service Specific Convergence Sublayer (SEG-SSCS) and the Common Part Sublayer (CPS). Amongst others the SEG-SSCS contains the Service Specific Segmentation and Reassembly (SSSAR) Sublayer. The SSSAR sublayer and the CPS are being investigated in this project, since the other are service specific and therefore they are not part of a generic SAR building block. The CPS can cooperate with multiple SSSAR entities in order to multiplex several data streams to provide the features explained above.

The AAL sublayers interchange data by passing messages. Those messages do not only contain the actual payload but they also contain the side information each layer needs for operation.

2.5.1 Service Specific Segmentation and Reassembly (SSSAR) sublayer

As stated in [4] the SSSAR transmitter accepts 1 through 65568 bytes of data – the *SSSAR Service Data Unit (SSSAR-SDU)* – from an upper layer, splits it appropriately into parts of 1 through 45 (default) or 1 through 64 bytes and hands the parts down to the CPS. Along with every part the SSSAR sublayer sends a *CPS-User-to-User-Indication (CPS-UUI)*. If the part currently being sent is the last part of the SSSAR-SDU, the CPS-UUI equals the *SSSAR-UUI* which the SSSAR sublayer

received together with the SSSAR-SDU. If the current part is not the last part of the SSSAR-SDU, the CPS-UUI equals 27 indicating that at least one more part is following.

Although [4] provides flow graphs showing exactly how SSSAR transmitters and receivers can be implemented, we generated new flow graphs. The reason for this is the following. Our goal is to develop a generic SAR building block being able to perform a number of protocols. Therefore, it is desirable to compare and, if possible, to unify the flow graphs of those protocols. A first step towards this goal is to create flow graphs of all protocols of the same level of detail. The RFCs describing PPP MUX and PPP MC contain only verbal descriptions of how those protocols could be implemented. The ITU-T Recommendations for the ATM Adaptation Layers on the other hand provide very detailed flow graphs. Thus a common abstraction level had to be found. The level we chose is low enough to clarify how the protocol works, but still high enough that a person using this document for implementing one or more protocols has enough freedom in the way he does it.

The flow graph in figure 5 illustrates the way an SSSAR transmitter works. Figure 6 shows the receiver. Rx operation starts from *IDLE*. Please consult [4] for more details.

2.5.2 Common Part Sublayer (CPS)

As stated in [2] the CPS transmitter accepts data of 1 through 45 (default) or 1 through 64 bytes length – the *CPS-SDU* – from the SSSAR sublayer. It assembles a so called *CPS-Packet* by prepending a 3 byte *CPS Packet Header (PH)* to the CPS-SDU. Amongst other things this header contains information about the length of the CPS-SDU: the *Length Indicator (LI)*. Multiple of those CPS-Packets can be concatenated. If the resulting data unit is too big for being transported in a single ATM cell it is split into multiple parts of exactly 47 bytes size. To be of 47 bytes size the last part is being padded if necessary. Another 1 byte header, called *Offset Field (OSF)*, is prepended to each one of those 47 byte parts. The OSF indicates where exactly in this part the first CPS-Packet is located. The resulting 48 byte *CPS Protocol Data Unit (CPS-PDU)* is passed to the ATM layer. Note, that since the transmitter is allowed to send data only after having received the respective permission from the Management Plane, queuing might be needed at the transmitter's input.

The CPS receiver gets the CPS-PDU from the ATM layer. It uses the information in the OSF to find the first CPS Packet Header and the information in each PH to reconstruct the CPS-Packets. Their payload is then passed to the SSSAR sublayer.

The flow graph in figure 7 illustrates the way a CPS transmitter works. Operation starts from *IDLE*. Figure 8 shows the receiver. Note that we introduced three states called *NORMAL*, *SPLIT* and *EXPECT*, which makes it easier to understand the way the receiver operates. Please consult [2] for more details.

2.6 ATM Adaptation Layer Type 5 (AAL 5)

AAL 5 is designed to accommodate variable bit rate data having no timing requirements from source to destination (e.g. classical IP traffic). Thus, AAL 5 offers a way to transmit long packets of data over an ATM network. To achieve its goals it adds a trailer conveying the information necessary for rebuilding the original packet, and aligns the resulting data unit to an integral multiple of 48 bytes by inserting padding. Since there cannot be more than one packet inside a single cell and thus the space remaining after having inserted a small packet is "wasted" by padding, using AAL 5 for transmitting small packets can be very bandwidth consuming. AAL 2 might be a better choice in case of small packets.

According to [3] AAL 5 consists of two sublayers: the Convergence Sublayer (CS), which is subdivided into the Service Specific Convergence Sublayer (SSCS) and the Common Part Convergence Sublayer (CPCS), and the Segmentation And Reassembly (SAR) Sublayer. This project investigates only the CPCS and the SAR Sublayer, since the SSCS is service specific and thus part of a higher layer.

2.6.1 Common Part Convergence Sublayer (CPCS)

CPCS has two modes of operation: *Message Mode* and *Streaming Mode*. Since according to [3] many issues about Streaming Mode operation are for further study, our documents considers only Message Mode operation, unless otherwise stated.

The CPCS transmitter accepts data of 1 through 65535 bytes length – the *CPCS-SDU* – and creates the *CPCS-PDU* out of it by padding it and adding a trailer. The padding aligns the overall CPCS-PDU to an integral multiple of 48 bytes. Amongst other things the trailer conveys information about the length of the CPCS-SDU. The CPCS-PDU is passed to the SAR sublayer.

Note that CPCS operation is not symmetric, which means that the CPCS receiver would not directly accept the data being sent by a CPCS transmitter. CPCS transmitter and receiver can only communicate with the SAR sublayer between them. The SAR transmitter splits data, but the SAR receiver does not perform reassembly. The reassembly is done by the CPCS receiver, although the CPCS transmitter does not split data.

The CPCS receiver accepts parts of 48 bytes size from the SAR sublayer. Along with this parts the CPCS receiver gets an M bit which indicates if the part currently received is the last one to complete the original CPCS-PDU or not. Upon completion, the original CPCS-SDU is extracted by using the length information in the trailer and passed to the CPCS user.

The flow graph in figure 9 illustrates the way a CPCS transmitter works. Figure 10 shows the receiver. Please consult [3] for more details.

2.6.2 Segmentation and Reassembly (SAR) sublayer

The SAR transmitter accepts data of 48 through 65568 bytes length (integral multiple of 48 bytes), splits it into parts of size 48 bytes and hands it to the ATM layer. The AUU bit sent with each part indicates whether this is the ending part of the CPCS-PDU or not.

The SAR receiver gets those 48 bytes of data and passes them to the CPCS without having altered them in any way. Along with this an M bit is sent which indicates if this part is the terminating one of the CPCS-PDU. It is set according to the received AUU value.

The flow graph in figure 11 illustrates the way a SAR transmitter works. Figure 12 shows the receiver. Please consult [3] for more details.

2.7 Inverse Multiplexing for ATM (IMA)

IMA's purpose is the transmission of ATM cells belonging to the same flow over multiple links and to retrieve the original cell stream at the far end. Thereby the bandwidth presented to the cell stream is increased. IMA introduces only very little overhead, which means that the bandwidth of the IMA bundle is nearly as high as the sum of the bandwidths of the bundle members. IMA works cell by cell, which means that it does not split or combine cells. It just makes sure that every cell goes to the appropriate link and that the cells are put into the right order again in the receiver. Preserving the order of the cells is done by inserting IMA Control Protocol (ICP) cells. IMA operation is completely transparent to ATM which means that ATM transmitter and receiver do not have to be aware of whether IMA is being performed. Since IMA does not perform segmentation and reassembly of cells, it is considered to be out of this document's scope.

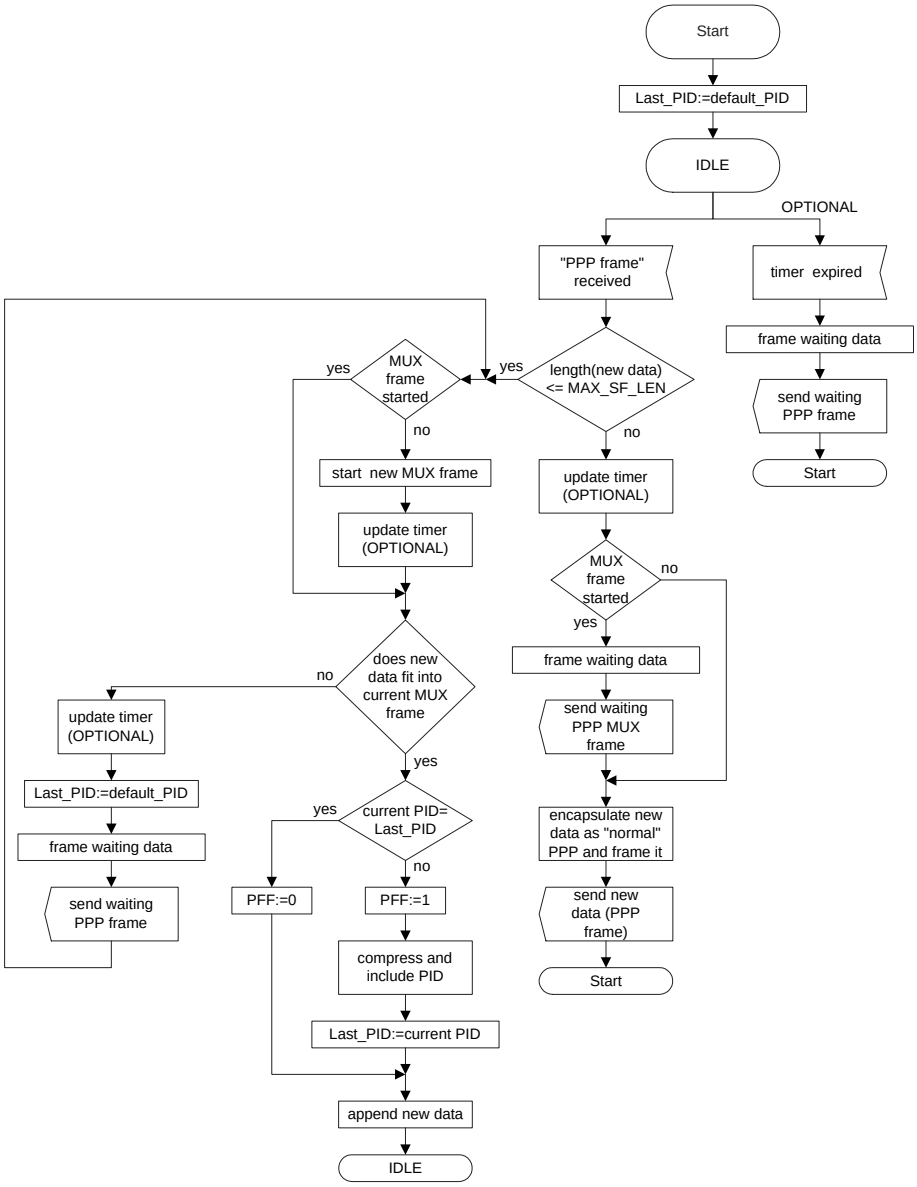

Figure 1: PPP MUX Tx

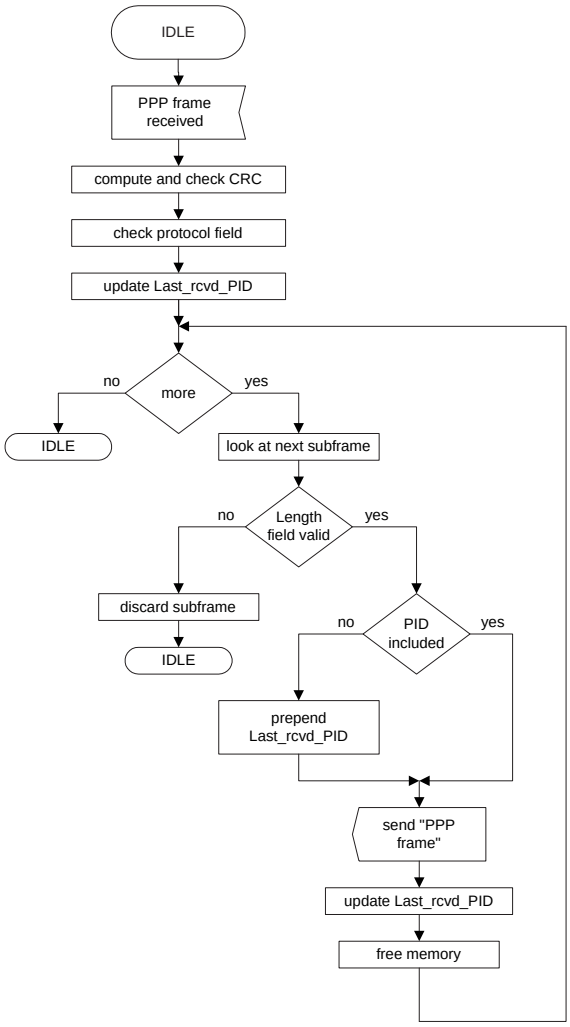

Figure 2: PPP MUX Rx

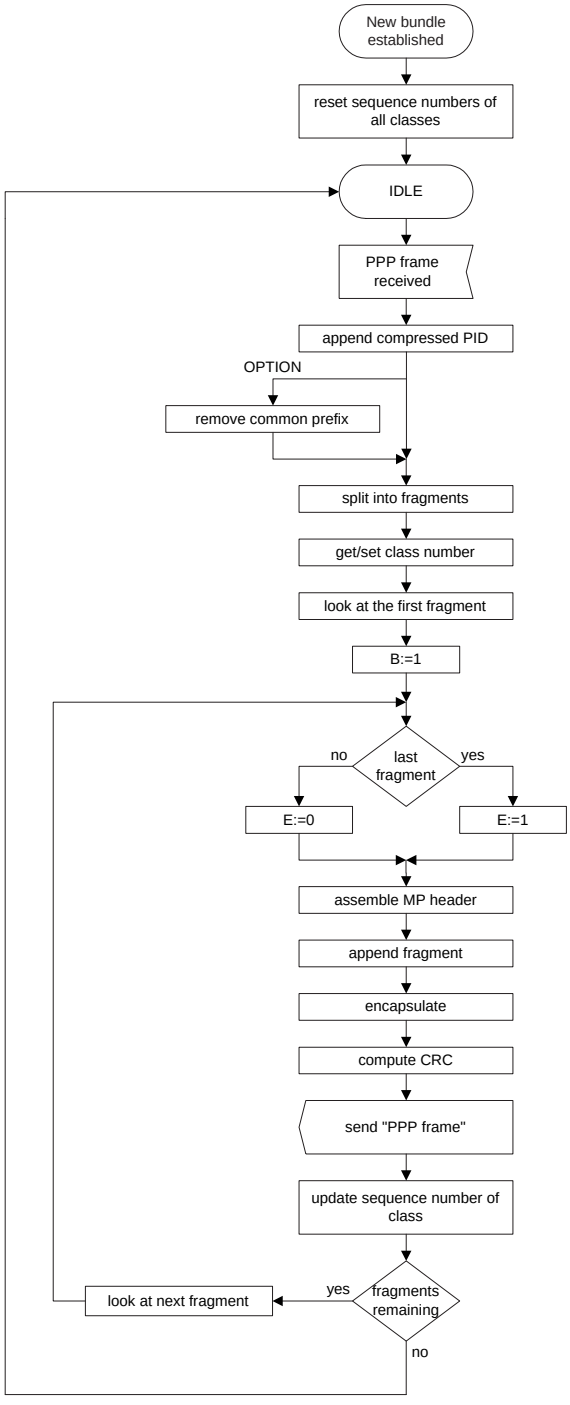

Figure 3: PPP MC Tx

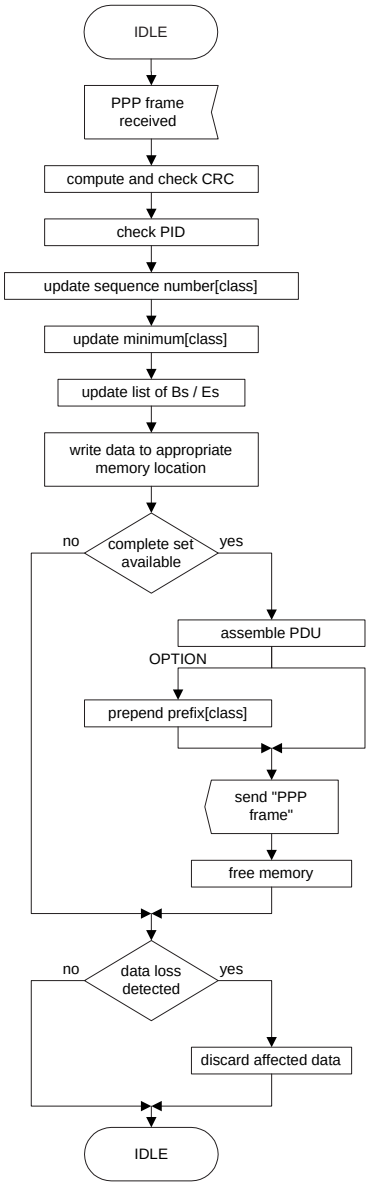

Figure 4: PPP MC Rx

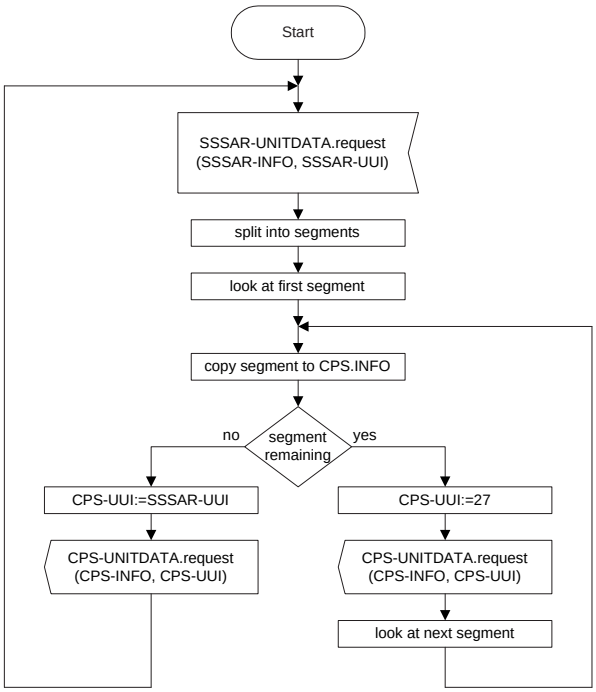

Figure 5: AAL2 SSSAR Tx

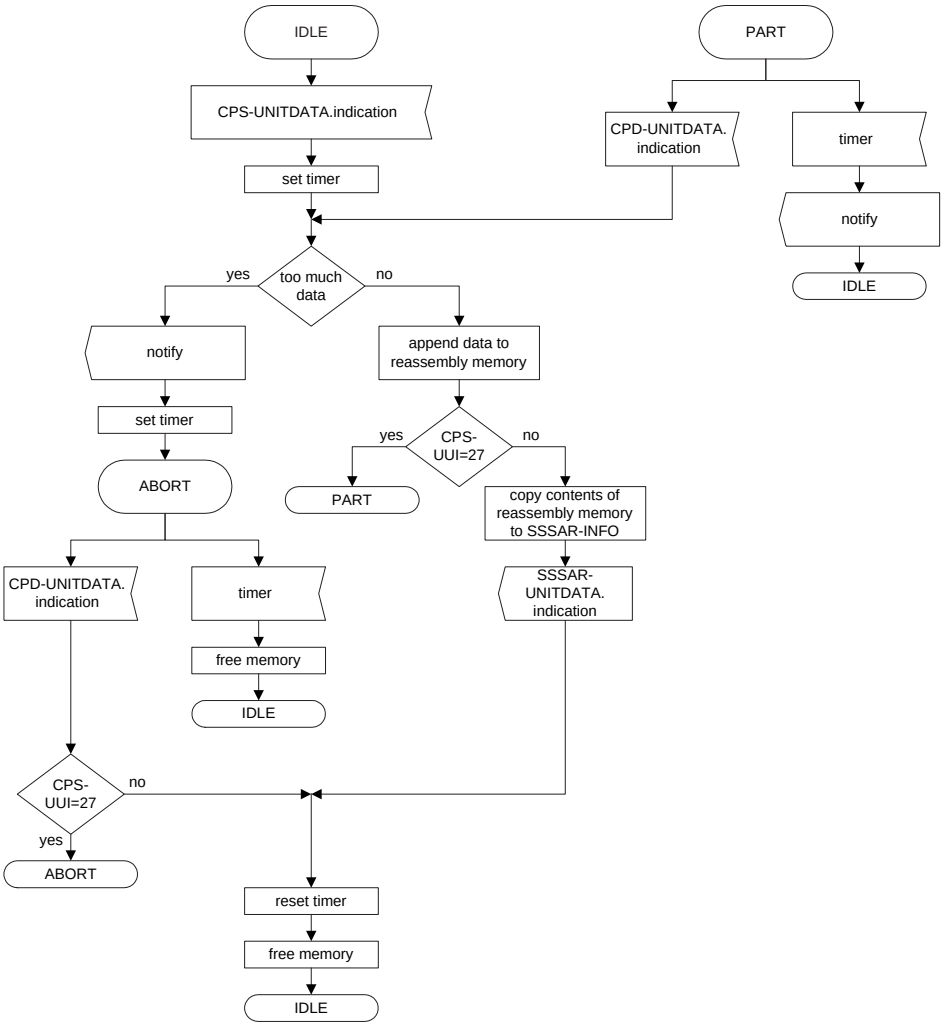

Figure 6: AAL2 SSSAR Rx

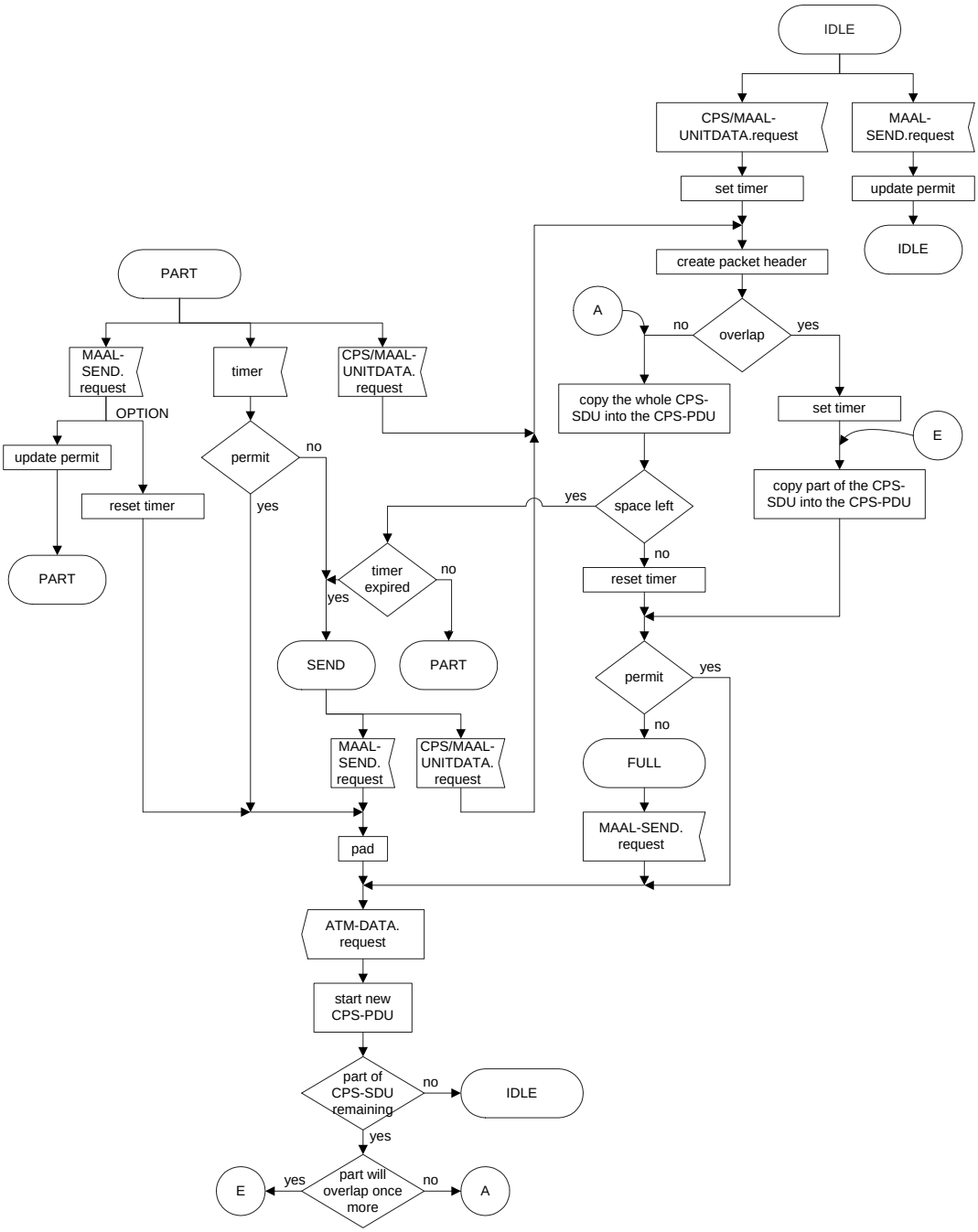

Figure 7: AAL2 CPS Tx

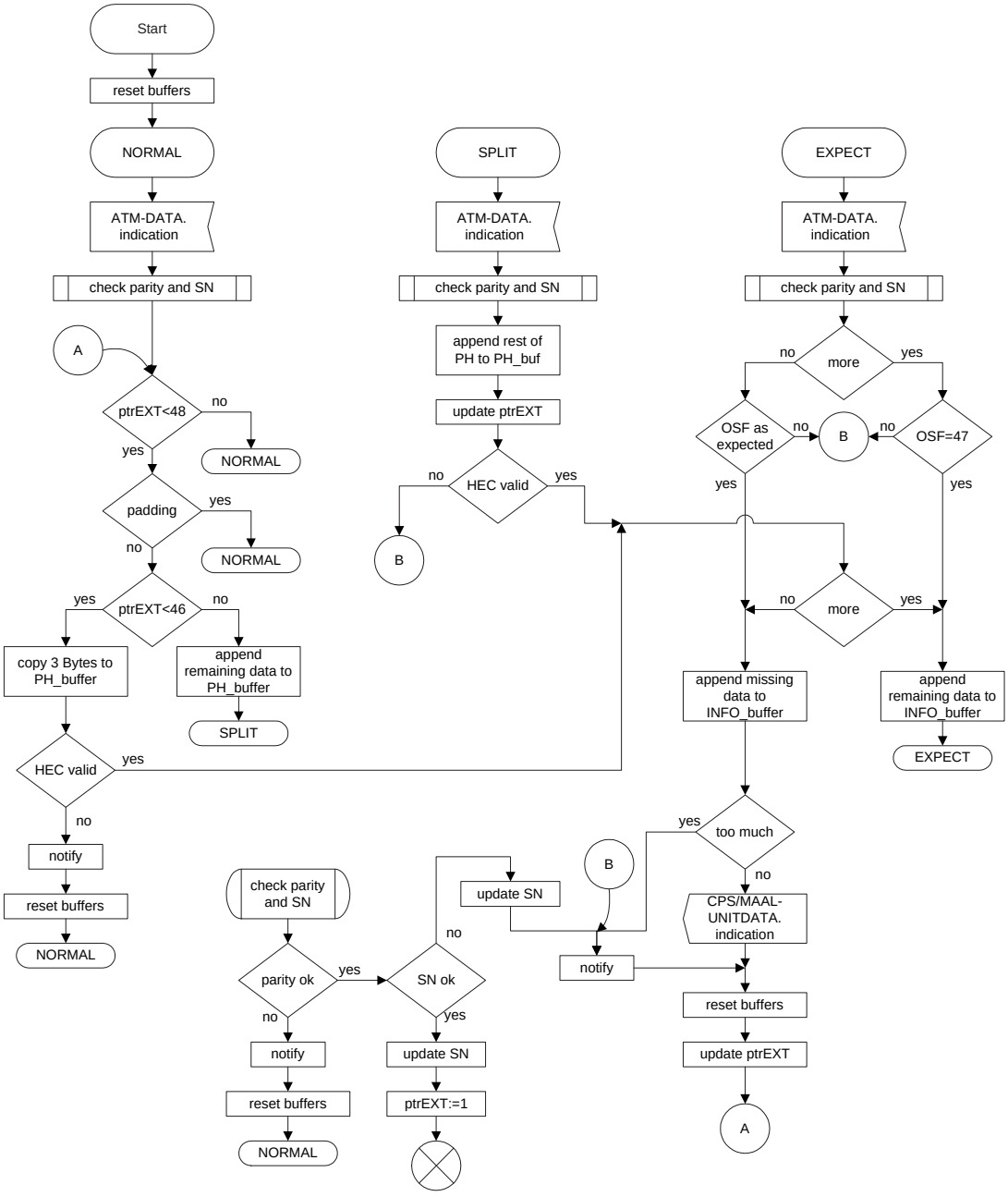

Figure 8: AAL2 CPS Rx

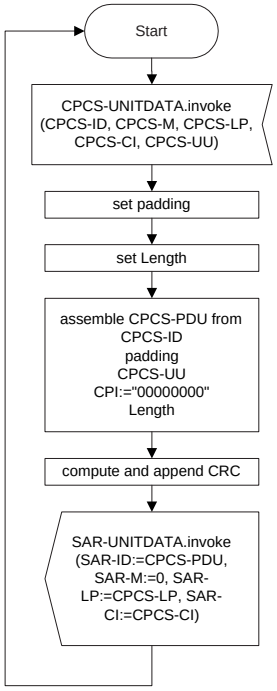

Figure 9: AAL5 CPCS Tx

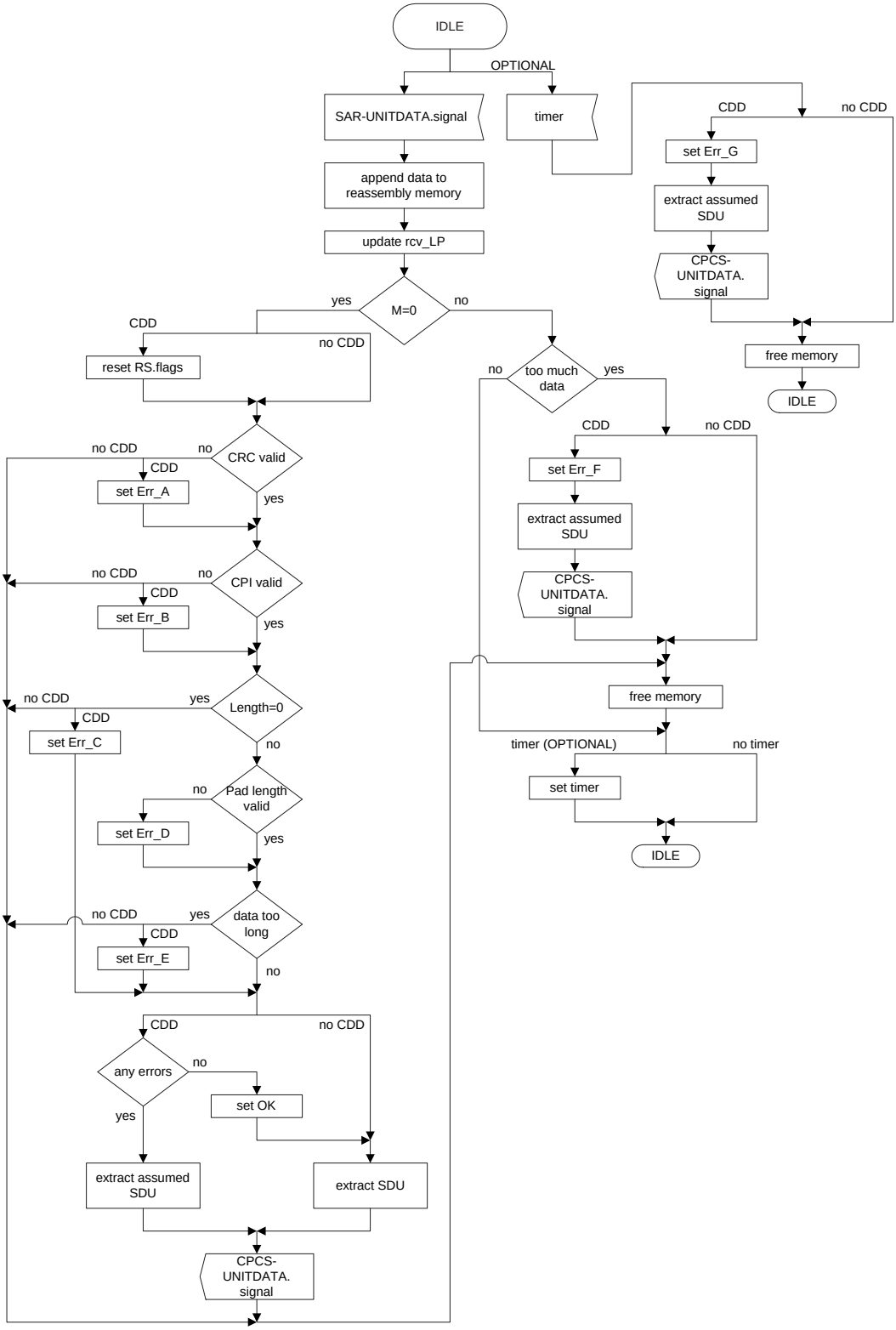

Figure 10: AAL5 CPCS Rx

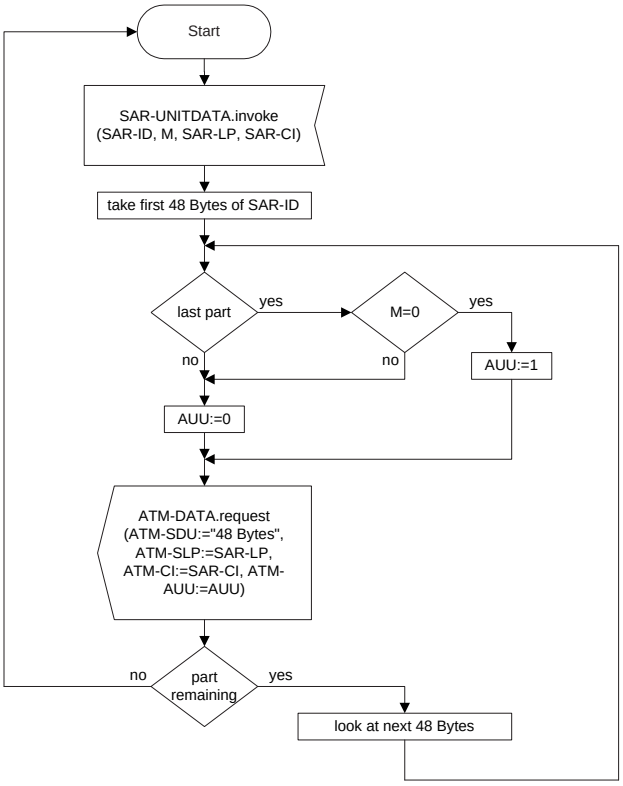

Figure 11: AAL5 SAR Tx

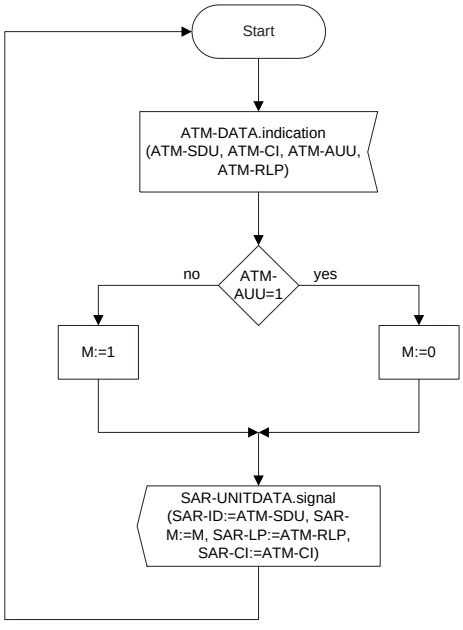

Figure 12: AAL5 SAR Rx

3 Composing a unified flow graph

Having derived flow graphs of equal abstraction levels for all protocols, we tried to combine several of them to find out to which extent their requirements in terms of functions and data flow would overlap. The findings we discovered while trying to create a generic transmitter and a generic receiver are described in the following.

3.1 The transmitter

Looking at the transmitters we found out that they can be classified as having either split or merge functionality. SSSAR, SAR and PPP MC can be considered as belonging to the splitting group, CPS and PPP MUX as belonging to the merging group. CPCS neither splits data units into several parts nor does it merge multiple data units into one. So an other way to decide whether to consider CPCS as one of the splitting or one of the merging protocols is to find the class which provides most of the functions being necessary to run CPCS. Amongst others CPCS requires one function that computes the length of a data unit and puts the result into one of the output data unit's fields, and an other function that adds padding to a data unit. Only CPS, which belongs to the merging group, needs the same two functions. Thus it seems reasonable to consider CPCS as one of the merging protocols.

Figure 13 illustrates how the unified transmitter works. Depending on the type of the input, data would flow through only one or through both the split and the merge block.

As one can see in figure 14 SSSAR, SAR and PPP MC can be combined very well. Compared to SSSAR and SAR, PPP MC needs some additional functions, but the order in which the common functions have to be performed is the same for each of the three protocols.

Trying to combine the merging protocols – CPS, CPCS and PPP MUX – we did not succeed. We did not find a way to draw a flow graph having many common parts. An example illustrating the problem is padding: performing CPCS, a data unit has to be padded in the very beginning of the flow, whereas in CPS, it has to be padded in the very end. What CPS and PPP MUX have in common is that both of them merge several data units into a single one. The big difference is that whenever PPP MUX gets an input data unit being bigger than the space remaining in the output data unit, the waiting data is sent and a new output data unit is begun. CPS first fills the waiting output data unit with as much as possible of the input data unit and then starts a new output data unit. Another difference is that PPP MUX sends its output data unit whenever it is filled or an optional timer expires, whereas CPS does not send any output data before it has got the permission from the Management Plane. There is no doubt that one can draw a flow graph covering CPS, CPCS and PPP MUX. But either it has many branches, which does not deliver any advantage compared to three single flow graphs, or several additional flags and checks have to be introduced, which means that the flow graphs being intended to show only the requirements of the protocols become more and more the illustration of a particular implementation.

3.2 The receiver

In Figures 15 and 16 one can find the attempts to combine the SSSAR and the CPCS receiver as well as the PPP MUX and the PPP MC receiver. A fundamental problem in combining several receivers is that the procedures for testing the validity of received data units are very protocol specific and vary to a great extent. Even if the protocols have some of the checks in common, their order differs in most of the cases. For instance Figure 15, which is abstracted to a high level, shows that in both SSSAR and CPCS it has to be checked whether more data is expected and whether too much data has already been received. However, SSSAR first checks, whether too many data has been received, whereas CPCS first checks, whether more data is expected. This difference makes it impossible to combine the two flow graphs without removing oneself too far from the actual requirements of the protocols.

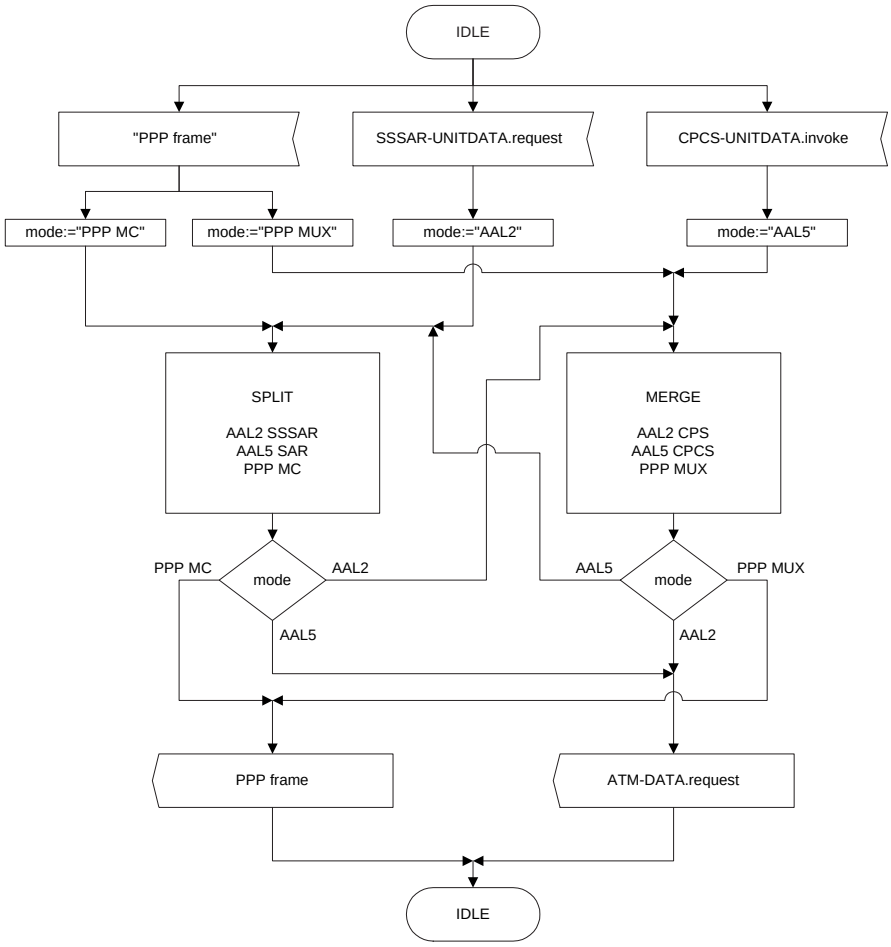

Figure 13: The unified transmitter

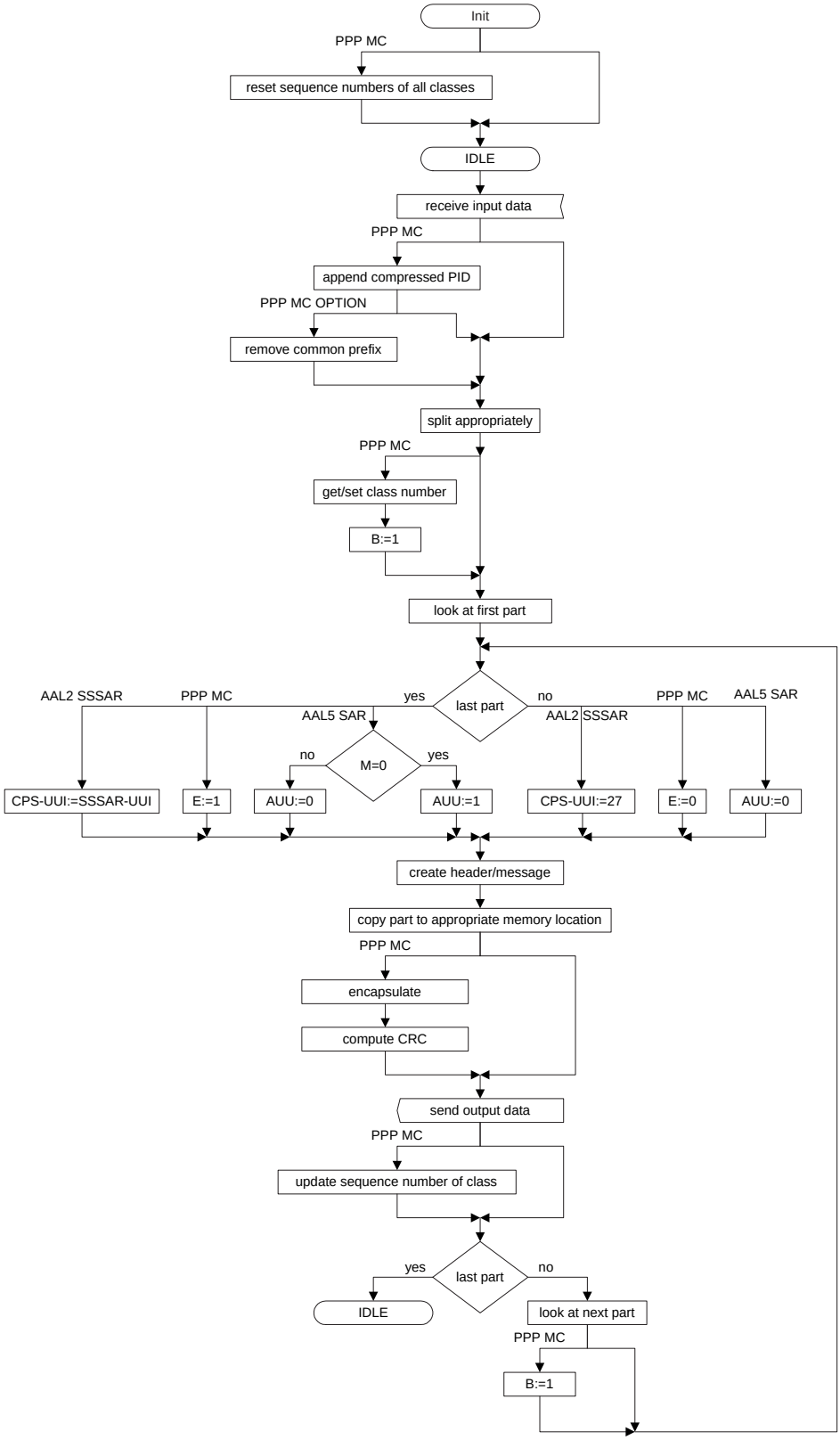

Figure 14: The splitting part of the transmitter

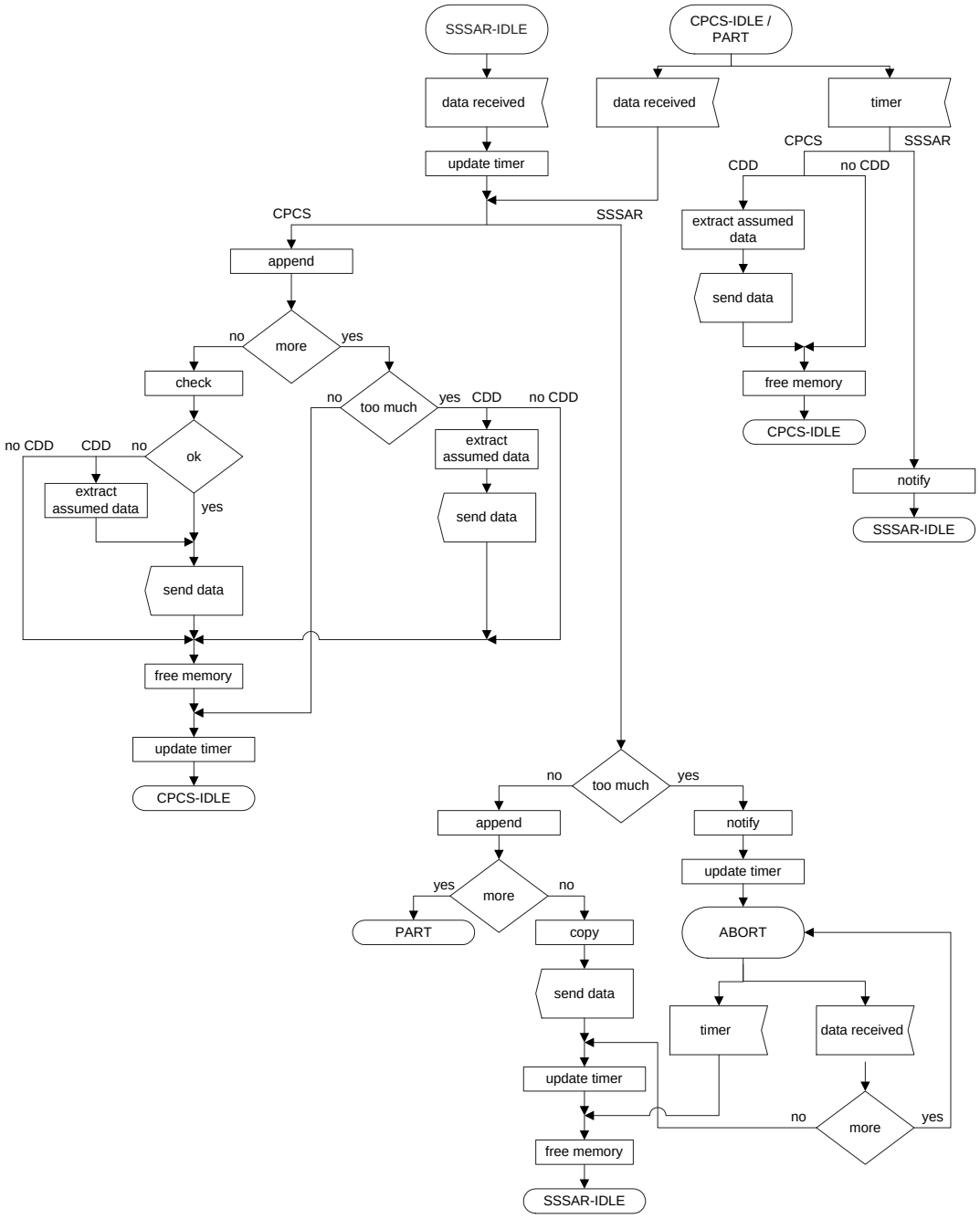

Figure 15: Combination of SSSAR Rx and CPCS Rx

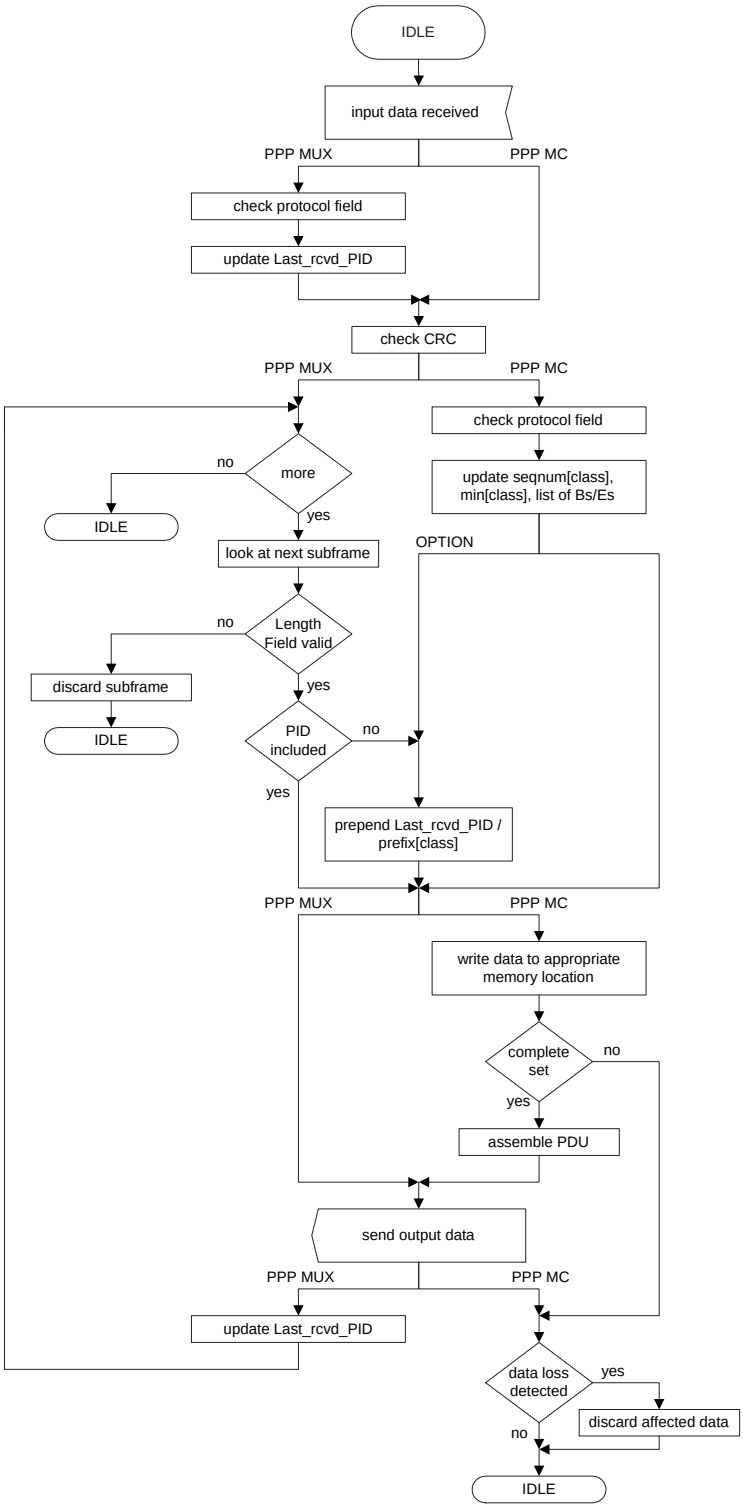

Figure 16: Combination of PPP MUX Rx and PPP MC Rx

4 Identifying the functional analogies

The first lesson we learned with this study is that the data flows vary from protocol to protocol. On the other hand many of the required operations are common. Therefore we now focus on the identification of the functional communalities among the different protocols without considering the data flow.

4.1 INMSGs and OUTMSGs

Since a part of the functions has to operate on data units, a generic format of each such data unit will be shown in the following. In general every protocol has to be able to get input data – even the transmitters – and to send output data – even the receivers. We mapped the data structures to a common skeleton. The input data structure is called input message (INMSG), the output data structure is called output message (OUTMSG). INMSGs and OUTMSGs generally consist of side information (SIDEINFO) and a data unit (DU). The DU is subdivided into a header (HDR), information (INFO) and a trailer (TRAILER). Table 1 shows the basic skeleton. Each one of the fields SIDEINFO, HDR, INFO and TRAILER can be subdivided into several fields. The way the side information is interchanged between layers is very implementation dependent and thus the given arrangement of several side information fields within the SIDEINFO and even the format of their content shall only be considered as an example. All the protocol specific fields are indicated by lower case letters whereas the skeleton fields are indicated by upper case letters.

INMSG / OUTMSG			
SIDEINFO	DU		
	HDR	INFO	TRAILER

Table 1: The message skeleton

A few general notes regarding INMSGs and OUTMSGs in transmitters and receivers: Since the generic SAR block is intended to be implemented in a system-on-a-chip (SoC) we assume that the format of the transmitters' INMSGs is very well defined and the SAR block is able to address each one of the INMSG's fields by its name at any instant of time after having got the INMSG. Addressing the fields by their names is also feasible in all the AAL receivers' INMSGs, because all of their fields have a fixed location within the INMSG.DU. This is not feasible in PPP MUX Rx and PPP MC Rx, because the only thing being fixed within those INMSG.DUs is their location relative to each others but not their absolute position. The sizes of several fields can vary from INMSG to INMSG. Therefore we printed their names in the respective tables in gray letters. The names can be used, but not before the appropriate locations have been identified – which we assume has to be done by the receiver itself. A similar problem exists with the OUTMSGs. Some of their fields' sizes may vary. Nevertheless we assume that their location is fixed and an implementation clears the "spaces" between the fields and also removes any unused fields before sending the data.

- **PPP MUX Tx**
  Table 2 shows the format of the INMSG and the OUTMSG being used by the PPP MUX transmitter.

  INMSG.DU.INFO.info carries the payload which – if not sent in a MUX frame – would have formed the Info Field of a normal PPP frame.

- **PPP MUX Rx**
  Table 3 shows the format of the INMSG and the OUTMSG being used by the PPP MUX receiver.

  OUTMSG.DU does not contain a complete PPP frame (i.e. with PPP header and a Frame Checksum). Since we consider the SAR block as being part of a SoC, it should be sufficient to have the PID (which can be found in the OUTMSG.SIDEINFO) and the Info Field. If it should still be necessary to create a valid PPP frame instead of only including the Info Field and the PID in the OUTMSG, this can easily be done by means of functions which are needed anyway for implementing the PPP MUX transmitter.

- **PPP MC Tx**
  Table 4 shows the format of the INMSG and the OUTMSG being used by the PPP MC transmitter.

  INMSG.DU.INFO.info carries the payload which – if not sent in a MC frame – would have formed the Info Field of a normal PPP frame.

  Note:
  [OUTMSG.DU.INFO.fragmentdata] $\leq$ min(MRU, MRRU) - 4 B if Multilink Short Sequence Number Header Option is not enabled
  [OUTMSG.DU.INFO.fragmentdata] $\leq$ min(MRU, MRRU) - 2 B if Multilink Short Sequence Number Header Option is enabled

- **PPP MC Rx**
  Table 5 shows the format of the INMSG and the OUTMSG being used by the PPP MC receiver.

  Additionally an implementation dependent intermediate structure, referred to as *intermediate_memory* in the respective table of functions, is needed.

  OUTMSG.DU does not contain a complete PPP frame (i.e. with PPP header and a Frame Checksum). Since we consider the SAR block as being part of a SoC, it should be sufficient to have the PID (which can be found in the OUTMSG.SIDEINFO) and the Info Field. If it should still be necessary to create a valid PPP frame instead of only including the Info Field and the PID in the OUTMSG, this can easily be done by means of functions which are needed anyway for implementing the PPP MC transmitter.

  Note:
  [INMSG.DU.INFO.fragmentdata] $\leq$ max(1500 B, min(MRU, MRRU)) - 4 B if Multilink Short Sequence Number Header Option is not enabled,
  [INMSG.DU.INFO.fragmentdata] $\leq$ max(1500 B, min(MRU, MRRU)) - 2 B if Multilink Short Sequence Number Header Option is enabled

- **AAL2 SSSAR Tx**
  Table 6 shows the format of the INMSG – SSSAR-UNITDATA.request – and the OUTMSG– CPS-UNITDATA.request – being used by the SSSAR transmitter.

- **AAL2 SSSAR Rx**
  Table 7 shows the format of the INMSG – CPS-UNITDATA.indication – and the OUTMSG – SSSAR-UNITDATA.indication – being used by the SSSAR receiver.

- **AAL2 CPS Tx**
  Table 8 shows the format of the INMSG – CPS-UNITDATA.request for transfer of user data or MAAL-UNITDATA.request for transfer of management data – and the OUTMSG – ATM-DATA.request – being used by the CPS transmitter. INMSG.SIDEINFO.cps-cid is only used in MAAL-UNITDATA.requests. It is not present in CPS-UNITDATA.requests.
  Table 8 also shows the intermediate structure *CPS-Packet* which we use to illustrate how the CPS transmitter works. Not every implementation will require this intermediate structure.

- **AAL2 CPS Rx**
  Table 9 shows the format of the INMSG – ATM-DATA.indication – and the OUTMSG – CPS-UNITDATA.indication for transfer of user data or MAAL-UNITDATA.indication for transfer of management data – being used by the CPS receiver.
  Table 9 also shows the intermediate structure *intermediate_hdr* which is used to temporarily store and verify CPS-Packet headers being split and transported in different INMSGs.

- **AAL5 CPCS Tx**
  Table 10 shows the format of the INMSG – CPCS-UNITDATA.invoke – and the OUTMSG – SAR-UNITDATA.invoke – being used by the CPCS transmitter.
  INMSG.SIDEINFO.m is only present in Streaming Mode Operation.

- **AAL5 CPCS Rx**
  Table 11 shows the format of the INMSG – SAR-UNITDATA.signal – and the OUTMSG – CPCS-UNITDATA.signal – being used by the CPCS receiver.
  Table 11 also shows two intermediate structures. *cpcs-pdu* is used to reconstruct the original CPCS-PDU. *rs* is used only in Message Mode operation (MM), and only if the Corrupted Data Delivery option (CDD) is enabled. The same applies for OUTMSG.SIDEINFO.rs. OUTMSG.SIDEINFO.m is only present in Streaming Mode Operation (SM).

- **AAL5 SAR Tx**
  Table 12 shows the format of the INMSG – SAR.UNITDATA.invoke – and the OUTMSG – ATM-DATA.request – being used by the SAR transmitter.

- **AAL5 SAR Rx**
  Table 13 shows the format of the INMSG – ATM-DATA.indication – and the OUTMSG – SAR-UNITDATA.signal – being used by the SAR receiver.

4.2 Functions

The functions needed for performing the protocols are divided into three groups: *configuration* functions, *data* functions and *data control* functions.

4.2.1 Configuration functions

There is only one single configuration function:

- **set**
  This function is mainly used to establish the parameters and variables which are necessary for the data functions and the data control functions during runtime. The actual values are assumed to be given by a control entity.

4.2.2 Data functions

Data functions are functions which manipulate data either originating from an INMSG, an OUTMSG or an intermediate structure, or being targeted to one of them. Data control functions are controlling how the data functions manipulate data, that is the data flow.

As mentioned earlier, the ITU-T Recommendations defining how the AAL sublayers work do this in a very low level way by giving sample implementations. Since not every action being done in those sample implementations is a real requirement to fulfil the standard, we abstracted from the low level as much as necessary to be able to find analogies between the different protocols.

A common property of all the data functions is that they are completely protocol independent. If one of those functions is implemented for one protocol it can be used for each one of the other protocols without having done any changes to the functions. In contrast to this many of the data control functions may be protocol dependent. This will be discussed later.

The data functions we identified are being explained in the following. For some of the functions a table is given, which contains the respective arguments and a few examples of how the functions can be used. The examples are taken from the tables in Appendix B, where the functions are applied to all protocols.

- **get**

get	INMSG

  get is used to make an INMSG available for further processing. Since the definition of a *get* function is highly correlated to the target system we decided to use a very generic description and let the details to the designer.

- **allocate_mem**

allocate_mem	*target*
allocate_mem	INMSG
allocate_mem	cps-packet
allocate_mem	OUTMSG

allocate_mem can be used to allocate memory for INMSGs, intermediate structures and OUTMSGs. Similar to the *get* function the use of this function is implementation dependent and therefore no special arguments are specified.

- **assemblev**

assemblev	*source*	*destination*	*quantity*
assemblev	"00000000"	OUTMSG.DU.TRAILER.cpi	1 B
assemblev	li	cps-packet.ph.li	6 b
assemblev	INMSG.SIDEINFO.sssar-uui	OUTMSG.SIDEINFO.cps-uui	5 b

assemblev puts data of the size *quantity*, counted from the least significant bit (LSB), from *source* to *destination*. Note that in all protocols the part selected by *quantity* is consistently taken from the LSB, which circumvents the requirement of a mask.

- **assemblep**

assemblep	source_location	destination_location	quantity
assemblep	&INMSG.DU.INFO.info + info_offset	&OUTMSG.DU.INFO.info	len
assemblep	&INMSG.DU.INFO.id	&cpcs-pdu + cpcs-pdu_offset	48 B

assemblep puts data of the size *quantity*, starting from *source_location*, to *destination_location*. Its main difference compared to *assemblev* is the size of data being assembled.

- **extractv**

extractv	source	destination	quantity
extractv	INMSG.DU.HDR.p	p	1 b
extractv	hdr_buffer.hec	hec	5 b
extractv	&INMSG.DU.INFO.payload + read_offset	pad_test	1 B

extractv puts data of the size *quantity* from *source* to *destination*. Counting the quantity starts from the most significant bit (MSB) of the value indicated by *source*.

It took us some time to identify the functions needed to move data from a source to a destination. It turned out that it is useful to have the functions that we call *assemblev*, *assemblep* and *extractv*. A difference between assemblev and extractv is that the requested quantity of data is taken from the LSB end and from the MSB end, respectively. Moreover, *assemblev* considers a well defined organization of the data structure it works on, while *extractv* works in data that have no clearly marked entries (e.g. input data buffers). The major difference between assemblev and extractv on the one hand and assemblep on the other hand is that when we use assemblep in one of the following tables we have to operate on a relative big quantity of data. Since in an implementation big amounts of data are often located in external memory and therefore tangible only by means of pointers, we use the "p" (from "pointer") in *assemblep* to indicate this. The "v" in *assemblev* and *extractv* indicates relative small quantity of data ("values"). Nevertheless extractv can have a pointer as an argument. "v" and "p" are meant to be hints to the reader of this document regarding the size of data an implementation has to deal with. One surely could combine those three functions into one. But in our opinion those three functions are a good choice in terms of illustrating the requirements and giving some hints to possible implications on an implementation at the same time.

- **computeLength**

computeLength	*input*	*destination*	*width of result*
computeLength	INMSG.DU.INFO.cps-info	li	6 b

computeLength computes the length (in bytes) of *input* and puts the result of the size *width-of-result* into *destination*.

- **computeCRC**

computeCRC	*input*	*destination*	*order*
computeCRC	cps-packet.ph.cid & cps-packet.ph.li & cps-packet.ph.uui	cps-packet.ph.hec	5

computeCRC computes the checksum over *input* and puts the result into *destination*. The order of the CRC polynomial is determined by *order*.

- **computeParity**

computeParity	*input*	*destination*
computeParity	OUTMSG.DU.HDR.osf & OUTMSG.DU.HDR.sn	OUTMSG.DU.HDR.p

computeParity computes the parity over *input* and puts the result into *destination*.

- **pad**

pad	*data unit*
pad	OUTMSG.DU.INFO.payload

pad appends a filler, which does not convey any information, to the *data unit* in order to align the resulting data unit to an integral multiple of 48 bytes. Note that only AAL2 CPS Tx and AAL5 CPCS Tx require padding. Since in CPS the padding bits have to be "0", whereas in CPCS any coding is allowed, a padding function adding "0" is applicable for both of them.

- **discard**

discard	*target*
discard	INMSG
discard	cps-packet
discard	OUTMSG

Some implementations might need a *discard* function in order to get rid of corrupted data. Since its usage is very implementation dependent we have not specified additional arguments.

- **send**

send	*OUTMSG*

send is used to send an OUTMSG. Similar to *get* we let the function definition details to the designer.

4.2.3 Data control functions

We make use of the following data control functions:

- **compare**

compare	*argument 1*	*argument 2*	*operator*
compare	pid	Last_PID	=
compare	pid	"0xC021"	=
compare	length(OUTMSG.DU.INFO) + length(INMSG.DU.INFO)	max_INFO_length	$\leq$
compare	"quantity of unprocessed data"	"0"	=

compare compares *argument 1* and *argument 2* in terms of the *operator*. As one can see in the given examples we use unprecise arguments like "quantity of unprocessed data" and a not specified length function in certain cases. Since our goal is to be as generic as possible, we try to illustrate what this function has to do but we do not answer the question how this has to be done. An answer to this question leading to an efficient implementation can not be given until the target platform is known. However, the requirements in terms of argument sizes have been identified for the well defined arguments, as will be described in chapter 5.

- **update**

update	*target*	NOTE
update	timer	Depending on protocol and implementation a timer has to be set to different values.
update	local.OUTMSG.DU.HDR. address.WIDTH	The width of the address field depends on its value. The *update* function has to keep track of this width.
update	Last_PID	Depending on the decision whether the protocol field is included in the Length Field of a PPP MUX subframe, Last_PID has to be kept up-to-date.
update	crc_order	The crc_order depends on the type of the received PPP frame. It has to be kept up-to-date by the *update* function.
update	&INMSG.DU.INFO. TRAILER.crc	&INMSG.DU.INFO.TRAILER.crc indicates the location of INMSG.DU.INFO.TRAILER.crc. The location has to be updated before an other function can address the INMSG.DU.INFO.TRAILER.crc by using its "name".
update	"list" of Bs and Es	The PPP MC receiver has to keep track of the sequence numbers of MC frames bearing a B or an E bit. How this "list" is implemented is up to the implementer.

As one can see from the notes in the above table *update* has to cover a wide range of functions. Even within one protocol *update* can be used for different tasks. *update* is actually a class of functions that might include addition, substraction, copy and even more complex forms of operations. What has to be done can be extracted from the notes in the respective table of each protocol in Appendix B or can be looked up in detail in the respective standard.

- **check**

check	*target*	NOTE
check	timer	Timer already expired?
check	"Complete set of fragments available?"	
check	"Fragments lost?"	
check	permit	Has the permission to send an OUTMSG already been received?

The *check* function is highly implementation dependent and its requirements vary from case to case. So it can be the assert of an interrupt (e.g. timer) or a substraction function (e.g. "Fragments lost?"). Its timing requirements may also vary but the study has not revealed any strict timing requirements. *check* has an "argument" giving only a hint to what has to be done. The details can be found in the respective standards and the tables of Appendix B.

- **split**

split	*data unit*
split	INMSG.DU.INFO.sssar-info

split is also a very implementation dependent function. Its main task is to decide in how many parts the *data unit* has to be split and where the boundaries of each part have to be located. This information could then be provided to an *update* function which maintains a read pointer. To ensure that every part is being processed, we use the function *LoopControl*.

- **LoopControl** is used in conjunction with *split* and takes care that all parts of a data unit that has been split by the *split* function are being processed without getting reordered.

- **increment**

increment	*variable*
increment	seq_num[class]

increment increases the value of the *variable* by 1.

- **decrement**

decrement	*variable*
decrement	li

decrement decreases the value of the *variable* by 1.

- **get_notification**
  enables an implementation to receive a notification sent by a control entity.

- **send_notification**
  can be used to notify a control unit about errors.

Tables 14 to 49 in Appendix B show those functions applied to each one of the studied protocols. The order in which the functions are listed does not necessarily correspond to the order in which they have to be used. The tables are distinguished into configuration, data and data flow for the transmit and the receive parts, respectively. They show the function name, the arguments and some notes for clarification.

4.3 Applying the functions to PPP MC Tx

The following C-like pseudo code illustrates the way one could use the above functions to map the flow graph of the PPP MC transmitter (Figure 3). Appendix B contains the functions that might be used for mapping the flow graphs of all protocols, but it does not contain information about the order in which the functions have to be applied.

Since the flow graphs were created in an early phase of this project, when the common functions were not yet known, the naming of functional blocks in the flow graphs may differ from the naming of the functions identified later. For this reason we put the names that can be found in the flow graphs as comments into the following code. The code shows how the functions might be used to map the functional blocks referred to in the comments.
To simplify the code we assume that all links over which PPP MC frames can be sent are of the same rate. Thus the split function can operate without distinguishing between different link rates and OUTMSG.SIDEINFO.linknumber does not have to be specified.
We chose a special notation for variables which refer to fields of INMSGs and OUTMSGs but are not part of the respective structures. An example is "local. OUTMSG.DU.INFO.fragmentdata.pid.WIDTH". This value contains the width of OUTMSG.DU.INFO.fragmentdata.pid. We could have used "pid.WIDTH" instead of our cumbersome choice, but as there are two PIDs (OUTMSG.DU.INFO.fragmentdata.pid and OUTMSG.DU.HDR.pid) which have to be clearly distinguished, we use the whole name of the item we refer to, prepend a "local." to indicate that the value is not part of the respective INMSG or OUTMSG, and append the description of the value stored in this variable, e.g. "WIDTH". To be consistent throughout the whole document we use this naming convention even if the danger of confusing two items is small.

```
void PPPMCTx (crc_order, class_min, class_max,
              local.OUTMSG.DU.HDR.address.WIDTH,
              local.OUTMSG.DU.HDR.control.WIDTH,
              local.OUTMSG.DU.INFO.mphdr.class.WIDTH,
              local.OUTMSG.DU.INFO.mphdr.zero.WIDTH,
              local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH)
    {
    SARClass        class;
    SARMSG          INMSG, OUTMSG;
    SARState        PPPMCState, TRUE=1;
    SARSN           seqnum[class_max-class_min+1];
    SARVAR          i, e, b, read_offset, write_offset, tmp,
                    fragmentlength, pid_highbyte, pid_highbyte_lsb,
                    local.INMSG.DU.INFO.LENGTH;
    SAR1bit         FALSE1b=0, TRUE1b=1;
    SAR2bit         FALSE2b=0;

    /* reset sequence numbers of all classes */
    for (i=class_min; i<=class_max; i++) seq_num[i]=0;

    PPPMCState = TRUE;
    while (PPPMCState)
        {
        /* IDLE */
        while ("No PPP frame in input memory") {}

        /* PPP frame received */
        get (INMSG);

        /* prepend compressed PID */
        update (local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH);
            /* The above  update  function could for example be
            implemented by using the following functions:

            extractv (INMSG.SIDEINFO.pid, pid_highbyte, sizeof(SARByte));
            assemblev (pid_highbyte, pid_highbyte_lsb, sizeof(SARbit));
            tmp = compare (pid_highbyte_lsb, FALSE1b, "=");
            if (tmp)
              local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH = sizeof(SARByte);
            else
              local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH = 2 * sizeof(SARByte);
            */

        local.INMSG.DU.INFO.LENGTH =
```

```
                local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH +
                length (INMSG.DU.INFO.info);
    /* The compressed PID is not yet prepended. Up to now only
    the length of INMSG.DU.INFO has been modified to consider
    the width of the PID */

/* O P T I O N A L: remove common prefix */
update (read_offset);
update (local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH);
    /* depending on the read_offset (a part of) the PID
    is included in the first OUTMSG.DU.INFO.fragmentdata */
assemblev (INMSG.SIDEINFO.pid, &OUTMSG.DU.INFO.fragmentdata,
           local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH);

/* split into fragments */
split (INMSG.DU.INFO.info);

/* get/set class number */
assemblev (INMSG.SIDEINFO.class, class, 4 * sizeof(SARbit));

/* look at the first fragment, that is assign the right value
to read_offset */
update (read_offset);

/* B:=1, since it is the first fragment */
update (b);

/* LoopControl, can be implemented as a  while  function */
while ("Unprocessed fragment remaining")
    {
    allocate_mem (OUTMSG);
        /* One could think about implementing the following
        variant
            PPPMCState = allocate_mem (OUTMSG);
        in order to terminate the while loop if the allocation
        of memory fails. This could also be useful in
        conjunction with other functions to handle certain
        error situations */

    tmp = check ("Is this the last fragment?");
    update (e);
        /* update (e), could be implemented as follows:
        if (tmp)
            {
            /* E:=1 */
```

```
        assemblev (TRUE1b, e, sizeof(SARbit));
        }
    else
        {
        /* E:=0 */
        assemblev (FALSE1b, e, sizeof(SARbit));
        } */

/* assemble MP header */
assemblev (b, OUTMSG.DU.INFO.mphdr.b, sizeof(SARbit));
assemblev (e, OUTMSG.DU.INFO.mphdr.e, sizeof(SARbit));
assemblev (class, OUTMSG.DU.INFO.mphdr.class,
           local.OUTMSG.DU.INFO.mphdr.class.WIDTH);
assemblev (FALSE2b, OUTMSG.DU.INFO.mphdr.zero,
           local.OUTMSG.DU.INFO.mphdr.zero.WIDTH);
assemblev (seq_num[class], OUTMSG.DU.INFO.mphdr.seqnum,
           local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH);
update (local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH);
assemblev (INMSG.SIDEINFO.class, OUTMSG.DU.INFO.mphdr.class,
           local.OUTMSG.DU.INFO.mphdr.class.WIDTH);

/* append fragment */
update (read_offset);
update (write_offset);
update (fragmentlength);
assemblep (&INMSG.DU.INFO.info + read_offset,
           &OUTMSG.DU.INFO.fragmentdata + write_offset,
           fragmentlength);

/* encapsulate */
update (local.OUTMSG.DU.HDR.pid.WIDTH);
    /* depending on whether PFC is negotiated */
assemblev ("0xFF", OUTMSG.DU.HDR.address,
           local.OUTMSG.DU.HDR.address.WIDTH);
assemblev ("0x03", OUTMSG.DU.HDR.control,
           local.OUTMSG.DU.HDR.control.WIDTH);
assemblev ("0x003D", OUTMSG.DU.HDR.pid,
           local.OUTMSG.DU.HDR.pid.WIDTH);

/* compute CRC */
computeCRC (OUTMSG.DU.HDR & OUTMSG.DU.INFO,
            OUTMSG.DU.TRAILER.crc, crc_order);

/* send PPP frame */
send (OUTMSG);
```

```
            /* update sequence number of class */
            increment (seq_num[class]);
                /* Care has to be taken that in case of an overflow
                the respective sequence number is being reset. */
            if (seq_num[class] > max_seq_num) seq_num[class] = 0;
                /* where max_seq_num is given by the width of the
                of the sequence number
                (local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH) */
            }
        }
    }
```

5 Requirements

Based on the analysis of chapters 2 to 4 we have identified the requirements of the protocols in terms of parameters, state variables, functions, timers, counters and memory requirements for working on data units. Particularly the memory requirements are implementation dependent. We identified the sizes of the INMSGs, intermediate structures and OUTMSGs of all protocols, but how much memory actually has to be allocated at a certain point of time depends on the implementation. In PPP MC one could for example allocate memory for all the OUTMSGs being needed to convey the parts of the INMSG.DU on reception of an INMSG. A less memory consuming implementation could for instance allocate memory for only one OUTMSG at one point of time. A protocol in which the memory having to be allocated for the INMSGs can vary to a great extent is the AAL2 CPS transmitter. Since an OUTMSG has to wait until the permit for its transmission has been given by the Management Plane, a strongly varying quantity of memory for holding the INMSGs may be needed.

The tables containing the requirements we identified can be found in Appendix C.

Table 52, representing the requirements for PPP MC Tx, is taken as an example to describe the contents of the tables in Appendix C.

Table "PARAMETERS" shows the parameters, the values they can take on and the size of the parameters. Some sizes may be implementation dependent. One reason for this is the following: In principle it is sufficient to use a single bit to store the information TRUE or FALSE. But as in a certain implementation it might be easier to use a whole byte to store this information instead of addressing a single bit, we let it up to the designer which sizes to use. The parameter PFC which conveys the information whether Protocol Field Compression was negotiated is one of those parameters with an implementation dependent size. The size of the parameter MAX_SF_LEN, indicating the maximum length in bytes that a subframe is allowed to be of, could also be considered as being implementation dependent, since the length could be measured in different units. But as the granularity of the lengths of most protocols' data units is one byte, it seems reasonable to take the byte as the reference unit and arrange the sizes of the parameters appropriately.

The biggest value that MAX_SF_LEN can take on is 16,383. 14 bits are sufficient to represent this value, so "14 b" is what can be found in the table. Additionally one can find a column specifying the entity setting the parameters, a column with notes and another one showing the respective references where further details can be looked up.

Table "STATE VARIABLES" shows the required state variables, the range of their values, their sizes, some notes and the respective references.

Table "FUNCTIONS" contains the functions we identified as being necessary, the size of the input they have to deal with and the size of their output. Where necessary some notes are given.

The same implementation dependencies as explained for the parameters hold also for the state variables and the functions.

Table "RESOURCES" contains the count of timers and counters needed. It is worth noting that our study did not reveal the requirement of any counters.

5.1 SAR's relative position in a network processor

One of the issues that we dealt with was the relative position of a SAR building block in a network processor (SoC) system. More specifically, its position in the transmit part of the system. Due to its functionality one certainly places it close to the ingress and egress ports (units) of the system. In the ingress side a SAR block is expected just after the receive unit of the system; the INMSG.DUs are received, identified as SARed, e.g. PPP MUX, and then sent to the SAR block so that the forwarding functions of the network processor (e.g. parsing, classification, lookup) are performed in the appropriate Protocol Data Units (OUTMSG.DUs).

The question is more challenging for the egress side due to the relation with the scheduling function. The scheduler's task is to decide which DU should be sent next, and depending on the system's capabilities and the traffic's requirements it can consider different queues (classes) and/or timing information (e.g. for ATM traffic). The above description implies that the location of the scheduler should be closer to the egress ports/queues. On the other hand, the SAR block prepares the DUs for transmission on the egress links, so one could argue that the location of SAR should be closer to the egress ports. In the following paragraphs we take a closer look on this topic per protocol.

The PPP MUX protocol multiplexes a number of INMSG.DUs that are assigned to the same port. In case of different classes, one could multiplex only INMSG.DUs of the same class in a single PPP MUX frame. However, a number of subframes may need to wait until the PPP MUX frame is full. Therefore, if no strict timing requirements exist, a PPP MUX function can be performed per port and/or queue and thus be located after the scheduler.

In the case of PPP MC it is more serviceable to have the scheduler after the SAR block because this allows higher priority PPP MC frames to be interleaved between lower priority PPP MC frames, enhancing the QoS capabilities of the system. The same message can be posed also for the PPP MP case, where packets of high priority are not split and can be interleaved between PPP MP frames.

In AAL2, cells are created by multiplexing a number of short INMSG.DUs. The end of the multiplexing process is determined by the combination of a timer and a permission given by the Management Plane. Considering that the AAL2 protocol has to fulfil certain timing requirements that are supported by the scheduler, we conclude that the location of the SAR block is advisable to be before the scheduler.

On the other hand, AAL5 has no timing requirements, and the cells are created by a single INMG.DU. Therefore, the location of the SAR block can be after the scheduler, which then works on the INMSG.DUs.

The AAL1 protocol has strict timing requirements, but since the SAR block basically in the transmit side only encapsulates the INMSG.DUs, the SAR block can be either before or after the scheduler.

The IMA function decides the type of cell, i.e. a regular or a filler cell, that should be inserted in which line, in order to perform cell rate decoupling. Therefore, this function should be part of the framer or transmit unit, and thus it should be after the scheduler. However in this document, the IMA function is not considered as being part of the SAR building block, since it takes ATM cells as input and forward them to the output without having changed them in any way.

To summarize the messages of the above paragraphs, the location of the SAR block seems to be more appropriate before the scheduler.

6 Summary

The goal of this internship was to identify the requirements of a generic SAR building block covering PPP MUX, PPP MC and the SAR related sublayers of AAL 2 and AAL 5. Studying the respective standards we encountered the different abstraction levels used in the RFCs (PPP) and the ITU-T Recommendations (AAL). In order to identify the requirements of a generic SAR building block in terms of functions and data flow we derived flow graphs of equal abstraction levels for all protocols and tried to combine several of them to find out to which extent they would overlap. Considering the transmitter we succeeded in combining those protocols which we classified as having splitting functionality. We did not succeed in combining the merging protocols. A reason for this is that for instance CPS – having merging functionality – even splits data units if necessary to efficiently use the space in a cell before merging several data units, whereas PPP MUX only merges integral data units. Combining the flow graphs of the receivers fails because of the protocol specific way of checking the validity of received data. Even if some of the checks are common among several protocols, the order in which they have to be performed is often different.

The first lesson we learned from this study is that the data flows vary from protocol to protocol. On the other hand many of the required operations are common. Therefore we focused on the identification of the functional communalities among the different protocols without considering the data flow.

In order to be able to operate on input and output data in a protocol independent way, we defined a message skeleton which is able to convey the data formats of all protocols. The mapping of the protocols' data units to this skeleton can be found

in Appendix A.

Identifying the functional requirements we classified them into configuration functions, data functions and data control functions. Data functions are implementation dependent but protocol independent. Data control functions are implementation dependent but may also be protocol dependent. Several of them, e.g. update, are actually classes of functions including for instance add, subtract, copy, etc. All functions applied to the protocols can be found in Appendix B.

Where it was possible to make statements on the quantitative requirements this was done. One can find the respective information in Appendix C.

Knowing that the protocols' input and output data can be unified in a common skeleton, that their functional requirements are very similar, but that the order in which those functions have to be applied varies to a great extent, one may conclude that an Application Specific Instruction set Processor (ASIP) is an appropriate platform for implementing a generic SAR building block.

6.1 Future work

One of the main drivers of this work was to abstract as much as possible from any implementation. Implementation details however will shape the definition of the data functions. Moreover, the data control functions would then need to be more well defined.

New SAR functions emerge, including the ADSL bonding and possibly the RTP mixer. Study of more protocols and the applicability of the functions that this work revealed is a worthy future work.

One of the issues that would be interesting to study are the forms of parallelism that can be accomplished. For example, the framing process requires a number of asseblev's that are data independent and can be performed in parallel.

Another interesting topic is to consider different memory allocation scenarios and how they do reflect to the operation and performance of assemblep.

Finally, an ASIP implementation that would implement the data functions and provide means to construct the data control functions would be of great interest.

References

[1] C. Bormann, RFC 2686, The Multi-Class Extension to Multi-Link PPP, September 1999

[2] International Telecommunication Union, ITU-T Recommendation I.363.2, B-ISDN ATM Adaptation Layer specification: Type 2 AAL, November 2000

[3] International Telecommunication Union, ITU-T Recommendation I.363.5, B-ISDN ATM Adaptation Layer specification: Type 5 AAL, August 1996

[4] International Telecommunication Union, ITU-T Recommendation I.366.1 Segmentation and Reassembly Service Specific Convergence Sublayer foer the AAL type 2, June 1998

[5] R. Pazhyannur et. al, RFC 3153, PPP Multiplexing, August 2001

[6] W. Simpson, RFC 1661, The Point-to-Point Protocol (PPP), July 1994

[7] K. Sklower et. al, RFC 1990, The PPP Multilink Protocol (MP), August 1996

A INMSGs and OUTMSGs

INMSG			
SIDEINFO	DU		
pid	HDR	INFO	TRAILER
2 B		**info**	
		≤ 65535 B (= max. value of MRU)	

OUTMSG								
SIDEINFO	DU							
	HDR			INFO				TRAILER
	address	**control**	**pid**	**subframe[i]**				**crc**
	0 / 1 B	0 / 1 B	1 / 2 B	**lengthfield**			**protocolfield**	**info**	0 / 2 / 4 B
	default: 1 B	default: 1 B	default: 2 B	**pff**	**lxt**	**len**	0 / 1 / 2 B	≤ MAX_SF_LEN	default: 2 B
				1 b	1 b	6 / 14 b			

Table 2: INMSG and OUTMSG PPP MUX Tx

INMSG					
SIDEINFO	DU				
	HDR			INFO	TRAILER
	address	**control**	**pid**	**info**	**crc**
	0 / 1 B	0 / 1 B	1 / 2 B	≤ max(1500 B, MRU)	0 / 2 / 4 B
	default: 1 B	default: 1 B	default: 2 B		default: 2 B

INMSG			
SIDEINFO	DU		
pid	HDR	INFO	TRAILER
2 B		**info**	
		≤ max(1500 B, MRU)	

Table 3: INMSG and OUTMSG PPP MUX Rx

INMSG											
SIDEINFO	DU										
class	**pid**	HDR	INFO								TRAILER
i.d.	2 B		**info**								
			≤ 65535 B (= max. value of MRRU)								

OUTMSG												
SIDEINFO	DU											
linknumber	HDR			INFO								TRAILER
i.d.	**address**	**control**	**pid**	**mphdr**							**fragmentdata**	**crc**
	0 / 1 B	0 / 1 B	1 / 2 B	**b**	**e**	**class**	**zero**	**seqnum**			see Note in 4.1	0 / 2 / 4 B
	default: 1 B	default: 1 B	default: 2 B	1 b	1 b	2 / 4 b	0 / 2 b	12 b / 3 B				default: 2 B
						default: 4 b	default: 2 b	default: 3 B				

Table 4: INMSG and OUTMSG PPP MC Tx

INMSG												
SIDEINFO	DU											
linknumber	HDR			INFO								TRAILER
i.d.	**address**	**control**	**pid**	**mphdr**							**fragmentdata**	**crc**
	0 / 1 B	0 / 1 B	1 / 2 B	**b**	**e**	**class**	**zero**	**seqnum**			see Note in 4.1	0 / 2 / 4 B
	default: 1 B	default: 1 B	default: 2 B	1 b	1 b	2 / 4 b	0 / 2 b	12 b / 3 B				default: 2 B
						default: 4 b	default: 2 b	default: 3 B				

intermediate_memory
i.d.

OUTMSG					
SIDEINFO	DU				
class	**pid**	HDR	INFO		TRAILER
i.d.	2 B		**info**		
			≤ 65535 B (= max. value of MRRU)		

Table 5: INMSG, intermediate structure and OUTMSG PPP MC Rx

INMSG			
SIDEINFO	DU		
sssar-uui	HDR	INFO	TRAILER
5 b		**sssar-info**	
		1 .. 65568 B	

OUTMSG			
SIDEINFO	DU		
cps-uui	HDR	INFO	TRAILER
5 b		**cps-info**	
		1 .. 45 / 64 B	
		default: 1 .. 45 B	

Table 6: INMSG and OUTMSG AAL 2 SSSAR Tx

INMSG			
SIDEINFO	DU		
cps-uui	HDR	INFO	TRAILER
5 b		**cps-info**	
		1 .. 45 / 64 B	
		default: 1 .. 45 B	

OUTMSG			
SIDEINFO	DU		
sssar-uui	HDR	INFO	TRAILER
5 b		**sssar-info**	
		1 .. 65568 B	

Table 7: INMSG and OUTMSG AAL 2 SSSAR Rx

INMSG				
SIDEINFO		DU		
cps-uui	**cps-cid**	HDR	INFO	TRAILER
5 b	1 B		**cps-info**	
			1 .. 45 / 64 B	
			default: 1 .. 45 B	

cps-packet				
ph				**pp**
cid	**li**	**uui**	**hec**	**info**
1 B	6 b	5 b	5 b	1 .. 45 / 64 B
				default: 1 .. 45 B

OUTMSG							
SIDEINFO			DU				
auu	**slp**	**ci**	HDR			INFO	TRAILER
1 b	1 b	1 b	**osf**	**sn**	**p**	**payload**	
		OPTIONAL	6 b	1 b	1 b	47 B	

Table 8: INMSG, intermediate structure and OUTMSG AAL 2 CPS Tx

INMSG							
SIDEINFO			DU				
auu	**rlp**	**ci**	HDR			INFO	TRAILER
1 b	1 b	1 b	**osf**	**sn**	**p**	**payload**	
			6 b	1 b	1 b	47 B	

intermediate_hdr			
cid	**li**	**uui**	**hec**
1 B	6 b	5 b	5 b

OUTMSG				
SIDEINFO		DU		
cps-uui	**cps-cid**	HDR	INFO	TRAILER
5 b	1 B		**cps-info**	
			1 .. 45 / 64 B	
			default: 1 .. 45 B	

Table 9: INMSG, intermediate structure and OUTMSG AAL 2 CPS Rx

INMSG						
SIDEINFO				DU		
m	**cpcs-lp**	**cpcs-ci**	**cpcs-uu**	HDR	INFO	TRAILER
1 b	1 b	1 b	1 B		**id**	
only in SM					$\leq$ 65535 B	

OUTMSG								
SIDEINFO			DU					
m	**sar-lp**	**sar-ci**	HDR	INFO	TRAILER			
1 b	1 b	1 b		**id**	**cpcs-uu**	**cpi**	**length**	**crc**
				$\leq$ 65560 B	1 B	1 B	2 B	4 B

Table 10: INMSG and OUTMSG AAL 5 CPCS Tx

INMSG					
SIDEINFO			DU		
m	**sar-lp**	**sar-ci**	HDR	INFO	TRAILER
1 b	1 b	1 b		**id**	
				48 B	

cpcs-pdu				
payload_and_padding	**trailer**			
$\leq$ 65568 B	**cpcs-uu**	**cpi**	**length**	**crc**
	1 B	1 B	2 B	4 B

rs										
flags								**val_a**	**val_b**	**val_c**
ok	**err_a**	**err_b**	**err_c**	**err_d**	**err_e**	**err_f**	**err_g**	1 B	2 B	4 B
1 b	1 b	1 b	1 b	1 b	1 b	1 b	1 b			

OUTMSG							
SIDEINFO					DU		
m	**cpcs-lp**	**cpcs-ci**	**cpcs-uu**	**rs**	HDR	INFO	TRAILER
1 b	1 b	1 b	1 B	8 B		**id**	
						$\leq$ 65535 B	

Table 11: INMSG, intermediate structure and OUTMSG AAL 5 CPCS Rx

INMSG					
SIDEINFO			DU		
m	**sar-lp**	**sar-ci**	HDR	INFO	TRAILER
1 b	1 b	1 b		**id**	
				$\leq$ 65568 B	

OUTMSG					
SIDEINFO			DU		
auu	**slp**	**ci**	HDR	INFO	TRAILER
1 b	1 b	1 b		**info**	
				48 B	

Table 12: INMSG and OUTMSG AAL 5 SAR Tx

INMSG					
SIDEINFO			DU		
auu	**rlp**	**ci**	HDR	INFO	TRAILER
1 b	1 b	1 b		**info**	
				48 B	

OUTMSG					
SIDEINFO			DU		
m	**sar-lp**	**sar-ci**	HDR	INFO	TRAILER
1 b	1 b	1 b		**id**	
				48 B	

Table 13: INMSG and OUTMSG AAL 5 SAR Rx

B Functions

The following pages contain the configuration, data and data control functions together with their arguments, as they could be applied to PPP MUX, PPP MC, AAL 2 and AAL 5. Note that the order in which the functions are listed does not necessarily correspond to the order in which they have to be performed.
The abbreviation *i.d.*, which can be found in several tables, means *implementation dependent*.
[x] means *size of x*.
update *&field* means update the location where *field* can be found.

Function	*Argument*	*Note*
set	MRU	default value: 1500 B
set	ACFC	default value: false
set	PFC	default value: false
set	default_PID	
set	MAX_SF_LEN	$\leq$ min(MRU-2, 16383 B)
set	max_INFO_length	OPTIONAL, can be used for comparison instead of MRU
set	crc_order	default: CCITT 16-bit FCS

Table 14: Configuration functions PPP MUX Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
extractv	INMSG.SIDEINFO.pid	pid	2 B	once per INMSG	needed to update Last_PID
assemblev	"0xFF"	OUTMSG.DU.HDR.address	local.OUTMSG.DU.HDR. address.WIDTH	once per OUTMSG	PPP header
assemblev	"0x03"	OUTMSG.DU.HDR.control	local.OUTMSG.DU.HDR. control.WIDTH	once per OUTMSG	PPP header
assemblev	"0x003D"	OUTMSG.DU.HDR.pid	local.OUTMSG.DU.HDR. pid.WIDTH	once per OUTMSG	identifies a PPP frame
assemblev	"0x0059"	OUTMSG.DU.HDR.pid	local.OUTMSG.DU.HDR. pid.WIDTH	once per OUTMSG	identifies a PPP MUX frame
assemblev	pid	OUTMSG.DU.HDR.pid	local.OUTMSG.DU.HDR. pid.WIDTH	once per OUTMSG	identifies an LCP frame
assemblev	pff	OUTMSG.DU.INFO.subframe[i]. hdr.lengthfield.pff	1 b	once per INMSG	indicates whether Protocol Field is included in the header of the subframe or not
assemblev	lxt	OUTMSG.DU.INFO.subframe[i]. hdr.lengthfield.lxt	1 b	once per INMSG	indicates width of OUTMSG.DU.INFO. subframe[i].lengthfield.len
computeLength	OUTMSG.DU.INFO.subframe[i]. protocolfield & OUTMSG.DU.INFO.subframe[i].info	OUTMSG.DU.INFO.subframe[i].hdr. lengthfield.len	local.OUTMSG.DU.INFO. subframe[i].lengthfield.len. WIDTH	once per INMSG	either 6 b or 14 b wide, depending on its value, related to lxt
assemblev	pid	OUTMSG.DU.INFO.subframe[i]. protocolfield	local.OUTMSG.DU.INFO. subframe[i].protocolfield. WIDTH	once per INMSG	Can be 1 or 2 B long, depending on whether pid can be compressed or not. Can also be 0, if protocolfield is not present in this subframe.
assemblep	&INMSG.DU.INFO	&OUTMSG.DU.INFO.subframe[i].info	length(INMSG.DU.INFO)	once per INMSG	
computeCRC	OUTMSG.DU.HDR & OUTMSG.DU.INFO	OUTMSG.DU.TRAILER.crc	crc_order	once per OUTMSG	mandatory: CCITT 16-bit FCS, OPTIONAL(FCS-Alternatives): Null FCS, CCITT 32-bit FCS
send	OUTMSG			once per OUTMSG	

Table 15: Data functions PPP MUX Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
check	timer				OPTIONAL
update	timer				OPTIONAL
update	local.OUTMSG.DU.HDR.address. WIDTH				OPTIONAL(ACFC), for LCP packets: always 1 B
update	local.OUTMSG.DU.HDR.control. WIDTH				OPTIONAL(ACFC), for LCP packets: always 1 B
update	local.OUTMSG.DU.HDR.pid.WIDTH				OPTIONAL(PFC), for LCP packets: always 2 B
update	Last_PID				If Protocol fields are to be omitted, Last_PID has to be updated. ("SHOULD" be done)
compare	pid	Last_PID	=	once per INMSG	for Protocol Field Compression, ("SHOULD" be done)
compare	pid	"0xC021"	=	once per INMSG	is it an LCP frame?
compare	pid	"0x0021"	≥	once per INMSG	for Protocol Field Compression, ("SHOULD" be done)
compare	pid	"0x00FD"	≤	once per INMSG	for Protocol Field Compression, ("SHOULD" be done)
update	local.OUTMSG.DU.INFO.subframe[i]. protocolfield.WIDTH				
compare	length(INMSG.DU.INFO)	MAX_SF_LEN	≤		to decide if input data can be included in a MUX frame or has to be sent in a "normal" PPP frame
compare	length(OUTMSG.DU.INFO) + length(INMSG.DU.INFO)	max_INFO_length	≤		max_INFO_length ≤ MRU. The maximum length of a PPP MUX frame may be chosen by the implementer. It must not be bigger than MRU. "17 b": sum can be 2*MRU
update	pff				intermediate value, to be included in lengthfield, indicates presence of Protocol Field in subframe
update	lxt				intermediate value, to be included in lengthfield, indicates size of lenengthfield.len
update	local.OUTMSG.DU.INFO.subframe[i]. hdr.lengthfield.len.WIDTH				the width can be either 6 b or 14 b
update	crc_order				OPTIONAL (FCS-Alternatives): the appropriate order of CRC has to be chosen depending on the type of data included in the frame (RFC 1570, p. 6)

Table 16: Data control functions PPP MUX Tx

Function	*Argument*	*Note*
set	MRU	default value: 1500 B
set	ACFC	default value: false
set	PFC	default value: false
set	default_PID	
set	crc_order	default: CCITT 16-bit FCS

Table 17: Configuration functions PPP MUX Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
extractv	INMSG.DU.HDR.pid	tmp_pid	2 B	once per INMSG	extract 2 B of assumed PID
extractv	tmp_pid	tmp_pid	1 B	$\leq$ once per INMSG	use only first Byte of PID
computeCRC	INMSG.DU.HDR & INMSG.DU.INFO	computed_crc	crc_order	once per INMSG	mandatory: CCITT 16-bit FCS, OPTIONAL(FCS-Alternatives): Null FCS, CCITT 32-bit FCS
extractv	INMSG.DU.TRAILER.crc	extracted_crc	local.INMSG.DU. TRAILER.crc.WIDTH	once per INMSG	in order to verify CRC
extractv	&INMSG.DU.INFO.info + pff_offset	pff	1 b	once per OUTMSG	the value of pff indicates the presence of the Protocol Field in the subframe
extractv	&INMSG.DU.INFO.info + lxt_offset	lxt	1 b	once per OUTMSG	the value of lxt indicates the width of the len field in the subframe
extractv	&INMSG.DU.INFO.info + len_offset	len	len.WIDTH	once per OUTMSG	the value of len indicates the length of the subframe's info field
extractv	&INMSG.DU.INFO.info + protocolfield_offset	protocolfield	2 B	once per OUTMSG	extract 2 B of assumed protocol field
extractv	protocolfield	protocolfield	1 B	$\leq$ once per INMSG	use only first Byte of protocolfield
assemblep	&INMSG.DU.INFO.info + info_offset	&OUTMSG.DU.INFO.info	len	once per OUTMSG	
assemblev	Last_rcvd_PID	OUTMSG.SIDEINFO.pid	2 B	once per OUTMSG	include PID in OUTMSG
discard				$\leq$ once per INMSG	
send	OUTMSG			once per OUTMSG	

Table 18: Data functions PPP MUX Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
compare	LSB(upper_byte(tmp_pid))	“1”	=		Is the PID 1 or 2 B long? (PFC)
compare	LSB(upper_byte(protocolfield))	“1”	=		Is the protocolfield (within a subframe) 1 or 2 B long? (PFC)
update	&INMSG.DU.INFO.TRAILER.crc				update the location of the CRC
update	crc_order				mandatory: CCITT 16-bit FCS, OPTIONAL(FCS-Alternatives): Null FCS, CCITT 32-bit FCS
update	local.INMSG.DU.TRAILER.crc. WIDTH				
compare	computed_crc	extracted_crc	=		verification of CRC
compare	tmp_pid	“0x0059”			Is it a PPP MUX frame?
update	pff_offset				the position of this field is not fixed in a frame
update	lxt_offset				the position of this field is not fixed in a frame
update	len_offset				the position of this field is not fixed in a frame
update	protocolfield_offset				the position of this field is not fixed in a frame
update	info_offset				the position of this field is not fixed in a frame
compare	lxt	“1”	=		to determine whether the len field is 6 b or 14 b wide
update	len.WIDTH				the len field can be 6 b or 14 b wide
compare	pff	“1”	=		to determine if the protocol field is present
update	Last_rcvd_PID				if a subframe without a protocol field has been received, Last_rcvd_PID is delivered alongside this subframe
compare	“quantity of unprocessed data”	“0”	=		If there is no more data to be processed, the demux process is stopped.
compare	“quantity of unprocessed data”	len	<		If the current len field is not valid, the last subframe is discarded.

Table 19: Data control functions PPP MUX Rx

Function	*Argument*	*Note*
set	MRU[link]	There might be different MRUs for each link. Default: 1500 B
set	MRRU	default: 1500 B
set	ACFC	default value: false
set	PFC	default value: false
set	crc_order	default: CCITT 16-bit FCS
set	local.OUTMSG.DU.INFO.mphdr.class.WIDTH	default: 4 b, OPTIONAL (Multilink Short Sequence Number Header Format): 2 b
set	local.OUTMSG.DU.INFO.mphdr.zero.WIDTH	default: 2 b, OPTIONAL (Multilink Short Sequence Number Header Format): 0 b
set	local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH	default: 3 B, OPTIONAL (Multilink Short Sequence Number Header Format): 12 b

Table 20: Configuration functions PPP MC Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
assemblev	"0xFF"	OUTMSG.DU.HDR.address	1 B	$\leq$ once per OUTMSG	part of PPP hdr, can be negotiated PER EACH LINK to be omitted (OPTION: ACFC)
assemblev	"0x03"	OUTMSG.DU.HDR.control	1 B	$\leq$ once per OUTMSG	part of PPP hdr, can be negotiated PER EACH LINK to be omitted (OPTION: ACFC)
assemblev	"0x003D"	OUTMSG.DU.HDR.pid	local.OUTMSG.DU.HDR.pid.WIDTH	once per OUTMSG	part of PPP hdr, default size: 2 B, can be negotiated PER EACH LINK to be only 1 B (OPTION: PFC)
assemblev	b	OUTMSG.DU.INFO.mphdr.b	1 b	once per OUTMSG	"b=1" indicates first fragment (_b_eginning)
assemblev	e	OUTMSG.DU.INFO.mphdr.e	1 b	once per OUTMSG	"e=1" indicates last fragment (_e_nd)
extractv	INMSG.SIDEINFO.class	class	i.d.	once per INMSG	
assemblev	INMSG.SIDEINFO.class	OUTMSG.DU.INFO.mphdr.class	local.OUTMSG.DU.INFO.mphdr.class.WIDTH	once per OUTMSG	assembly of MP header
assemblev	"00"	OUTMSG.DU.INFO.mphdr.zero	local.OUTMSG.DU.INFO.mphdr.zero.WIDTH	once per OUTMSG	assembly of MP header
assemblev	seq_num[class]	OUTMSG.DU.INFO.mphdr.seqnum	local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH	once per OUTMSG	assembly of MP header
assemblev	INMSG.SIDEINFO.pid	&OUTMSG.DU.INFO.fragmentdata + write_offset	local.OUTMSG.DU.INFO.fragmentdata.pid.WIDTH	once per INMSG	Depending on its value the received PID is included (after PFC) into the OUTMSG.DU.INFO.fragmentdata
assemblep	&INMSG.DU.INFO.info + read_offset	&OUTMSG.DU.INFO.fragmentdata + write_offset	fragmentlength	once per OUTMSG	"read_offset" considers both removing of common prefix (OPTIONAL) and fragment boundaries
computeCRC	OUTMSG.DU.HDR & OUTMSG.DU.INFO	OUTMSG.DU.TRAILER.crc	crc_order	once per OUTMSG	mandatory: CCITT 16-bit FCS, OPTIONAL(FCS-Alternatives): Null FCS, CCITT 32-bit FCS
send	OUTMSG			once per OUTMSG	

Table 21: Data functions PPP MC Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
split	INMSG.DU.INFO.info				split into appropriate fragments, possible strategies see RFC 1717, p. 7
LoopControl					
update	fragmentlength				up to MRU - 2 B
update	b				"b=1" indicates first fragment (_b_eginning)
update	e				"e=1" indicates last fragment (_e_nd)
update	local.OUTMSG.DU.HDR.address. WIDTH				OPTIONAL(ACFC) per each link
update	local.OUTMSG.DU.HDR.control. WIDTH				OPTIONAL(ACFC) per each link
update	local.OUTMSG.DU.HDR.pid.WIDTH				OPTIONAL(PFC) per each link
update	local.OUTMSG.DU.INFO. fragmentdata.pid.WIDTH				determines the width of the PID that might be included in the first fragment
update	crc_order				mandatory: CCITT 16-bit FCS, OPTIONAL(FCS-Alternatives): Null FCS, CCITT 32-bit FCS
update	read_offset				for both removing of common prefix (OPTIONAL) and fragment boundaries
update	write_offset				The location within an OUTMSG where "real data" begins depends on the length of the PID.
update	OUTMSG.SIDEINFO.linknumber				maybe an other part of the system makes the decision over which link the frame shall be sent
increment	seq_num[class]				

Table 22: Data control functions PPP MC Tx

Function	*Argument*	*Note*
set	ACFC	default value: false
set	PFC	default value: false
set	crc_order	default: CCITT 16-bit FCS
set	local.OUTMSG.DU.INFO.mphdr.class.WIDTH	default: 4 b, OPTIONAL (Multilink Short Sequence Number Header Format): 2 b
set	local.OUTMSG.DU.INFO.mphdr.zero.WIDTH	default: 2 b, OPTIONAL (Multilink Short Sequence Number Header Format): 0 b
set	local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH	default: 3 B, OPTIONAL (Multilink Short Sequence Number Header Format): 12 b
set	prefix[class]	OPTIONAL (Prefix Elision), every class can have its own prefix, but there can be one common prefix for all classes, too (RFC 2686, p. 8)

Table 23: Configuration functions PPP MC Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
extractv	INMSG.DU.HDR.pid	tmp_pid	2 B	once per INMSG	extract 2 B of assumed PID
extractv	tmp_pid	tmp_pid	1 B	$\leq$ once per INMSG	use only first Byte of PID
computeCRC	INMSG.DU.HDR & INMSG.DU.INFO	computed_crc	crc_order	once per INMSG	mandatory: CCITT 16-bit FCS, OPTIONAL(FCS-Alternatives): Null FCS, CCITT 32-bit FCS
extractv	INMSG.DU.TRAILER.crc	extracted_crc	local.INMSG.DU. TRAILER.crc.WIDTH	once per INMSG	in order to verify CRC
extractv	INMSG.DU.INFO.mphdr.b	b	1 b	once per INMSG	"b=1" indicates first fragment (_b_eginning)
extractv	INMSG.DU.INFO.mphdr.e	e	1 b	once per INMSG	"e=1" indicates last fragment (_e_nd)
extractv	INMSG.DU.INFO.mphdr.class	class	2 b / 4 b	once per INMSG	
extractv	INMSG.DU.INFO.mphdr.seqnum	sequencenumber[class]	12 b / 3 B	once per INMSG	
assemblep	&INMSG.DU.INFO.fragmentdata	&intermediate_memory + intermediate_memory_offset	length(INMSG.DU.INFO. fragmentdata)	once per INMSG	
assemblep	&prefix[class]	&OUTMSG.DU.INFO.info	length(prefix[class])	once per OUTMSG	OPTIONAL, prepend the prefix
assemblep	&intermediate_memory + intermediate_memory_offset	&OUTMSG.DU.INFO.info + write_offset	assemble_length	once per INMSG	reassembly of the original frame.info
discard					
send	OUTMSG			once per OUTMSG	

Table 24: Data functions PPP MC Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
compare	LSB(upper_byte(tmp_pid))	"1"	=		Is the PID 1 or 2 B long? (PFC)
update	&INMSG.DU.INFO.TRAILER.crc				update the location of the CRC
update	crc_order				mandatory: CCITT 16-bit FCS, OPTIONAL(FCS-Alternatives): Null FCS, CCITT 32-bit FCS
update	local.INMSG.DU.TRAILER.crc. WIDTH				
compare	computed_crc	extracted_crc	=		verification of CRC
compare	tmp_pid	"0x3D"			Is it a PPP MP/MC frame?
update	sequencenumber[class]				update the sequence number of the current INMSG's class
update	minimum[class]				maintain the current minimum of the most recently received sequence number over all links (RFC 1990, p.11)
update	"list" of Bs and Es				keep track of INMSGs with the b or e bit set
update	&INMSG.DU.INFO				depending on width of INMSG.DU.HDR.pid
update	&INMSG.DU.INFO.TRAILER				depending on width of INMSG.DU.HDR.pid
check	complete set of fragments available?				
update	intermediate_memory_offset				Needed for writing to the intermediate_memory and for reading from it. Has to be updated in such a way that the fragments are in the right order when finally assembled.
update	write_offset				Has to be updated in such a way that the fragments are in the right order when finally assembled.
update	assemble_length				indicates the size of the fragments during reassembly
check	fragments lost?				

Table 25: Data control functions PPP MC Rx

Function	*Argument*	*Note*
set	cps-info.MAXSIZE	can be either 45 B or OPTIONAL 64 B

Table 26: Configuration functions AAL2 SSSAR Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
assemblep	&INMSG.DU.INFO.sssar-info	&OUTMSG.DU.INFO.cps-info	cps-info.MAXSIZE	once per INMSG	take a part of the received data and put it into the OUTMSG
assemblev	"27"	OUTMSG.SIDEINFO.cps-uui	5 b	once per INMSG	"27" indicates that there's at least one more part missing
assemblev	INMSG.SIDEINFO.sssar-uui	OUTMSG.SIDEINFO.cps-uui	5 b	for the last part of a sssar-info	a value $\neq$27 marks the last part
send	OUTMSG			once per OUTMSG	

Table 27: Data functions AAL2 SSSAR Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
split	INMSG.DU.INFO.sssar-info				The size of each single part can be defined by the implementer. It has to be smaller than 45 B (OPTIONAL: 64 B)
LoopControl					
check	last part of INMSG.DU.INFO.sssar-info?				The value of OUTMSG.SIDEINFO.cps-uui depends on whether the part currently processed is the last one of a INMSG.DU.INFO.sssar-info or not

Table 28: Data control functions AAL2 SSSAR Tx

Function	*Argument*	*Note*
set	Max_SDU_Length	default: 65568 B, can be smaller
set	cps-info.MAXSIZE	can be either 45 B or OPTIONAL 64 B

Table 29: Configuration functions AAL2 SSSAR Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
assemblep	&INMSG.DU.INFO.cps-info	&OUTMSG.DU.INFO.cps-info + write_offset	length(INMSG.DU.INFO. cps-info)	once per INMSG	append INMSG.DU.INFO. cps-info to the OUTMSG
extractv	INMSG.SIDEINFO.cps-uui	cps-uui	5 b	once per INMSG	to determine whether there's at least on more part missing
assemblev	cps-uui	OUTMSG.SIDEINFO.sssar-uui	5 b	once per OUTMSG	
discard					
send	OUTMSG			once per OUTMSG	

Table 30: Data functions AAL2 SSSAR Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
update	timer				
check	timer				
update	write_offset				needed to append the INMSG.DU.INFO.cps-info to the right position of the OUTMSG
compare	cps-uui	"27"	=		Last part?
compare	length(OUTMSG.DU.INFO. sssar-info) + length(INMSG.DU.INFO.cps-info)	Max_SDU_Length	$\leq$		Is the data that has been received too long?
send_notification					notify the management plane about errors

Table 31: Data control functions AAL2 SSSAR Rx

Function	*Argument*	*Note*
set	cps-info.MAXSIZE	can be either 45 B or OPTIONAL 64 B

Table 32: Configuration functions AAL2 CPS Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
assemblev	snd_CID	cps-packet.ph.cid	1 B	once per INMSG	snd_CID is associated with the Connection End-point Suffix of the Access Point through which the CPS-UNITDATA.request primitive was received
computeLength	INMSG.DU.INFO.cps-info	li	6 b	once per INMSG	
decrement	li	1		once per INMSG	
assemblev	li	cps-packet.ph.li	6 b	once per INMSG	The LI field is binary encoded with a value one less than the number of bytes in the INMSG.DU.INFO.cps-info
assemblev	INMSG.SIDEINFO.cps-uui	cps-packet.ph.uui	5 b	once per INMSG	mapped from INMSG to cps-packet
computeCRC	cps-packet.ph.cid & cps-packet.ph.li & cps-packet.ph.uui	cps-packet.ph.hec	order = 5	once per INMSG	
assemblep	&INMSG.DU.INFO.cps-info	&cps-packet.pp.info	length(INMSG.DU.INFO. cps-info)	once per INMSG	mapping of input data into the intermediate structure
assemblep	&cps-packet + read_offset	&OUTMSG.DU.INFO.payload + write_offset	quantity	$\leq$ 3 times per INMSG	mapping parts of the CPS-Packet into the OUTMSG
pad	OUTMSG.DU.INFO.payload			$\leq$ once per OUTMSG	0 to 47 Byte ('0's)
assemblev	"0"	OUTMSG.SIDEINFO.auu	1 b	once per OUTMSG	always "0"
assemblev	"0"	OUTMSG.SIDEINFO.slp	1 b	once per OUTMSG	always "0"
assemblev	"0"	OUTMSG.SIDEINFO.ci	1 b	once per OUTMSG	always "0"
assemblev	osf	OUTMSG.DU.HDR.osf	6 b	once per OUTMSG	
assemblev	sn	OUTMSG.DU.HDR.sn	1 b	once per OUTMSG	
computeParity	OUTMSG.DU.HDR.osf & OUTMSG.DU.HDR.sn	OUTMSG.DU.HDR.p		once per OUTMSG	see ITU-T I.363.2 (11/2000), p. 11
send	OUTMSG			once per OUTMSG	

Table 33: Data functions AAL2 CPS Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
update	timer				needed to determine when to send an OUTMSG
check	timer				needed to determine when to send an OUTMSG
update	snd_CID				snd_CID is associated with the Connection End-point Suffix of the Access Point through which the CPS-UNITDATA.request primitive was received
update	osf				indicates the beginning of a new CPS-Packet within an INMSG.DU.INFO.payload
update	sn				compute sequence number
get_notification					receive MAAL-SEND.request
update	permit				on reception of MAAL-SEND.request
check	permit				needed to determine when to send an OUTMSG
update	read_offset				current position in cps-packet
update	write_offset				current position in payload of OUTMSG
update	quantity				

Table 34: Data control functions AAL2 CPS Tx

Function	*Argument*	*Note*
set	Max_CPS-SDU_Length	can be either 45 B or OPTIONAL 64 B
set	Max_SDU_Deliver_Length	can be either 45 B or OPTIONAL 64 B

Table 35: Configuration functions AAL2 CPS Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
extractv	INMSG.DU.HDR.p	p	1 b	once per INMSG	extracting parity bit for parity check
discard					depending on error situation
extractv	INMSG.DU.HDR.sn	sn	1 b	once per INMSG	extracting sequence number for sequence number check
extractv	INMSG.DU.HDR.osf	osf	6 b	once per INMSG	extracting offset field in order to determine location of certain data in INMSG
computeParity	osf & sn	parity		once per INMSG	see ITU-T I.363.2 (11/2000), p. 11
extractv	&INMSG.DU.INFO.payload + read_offset	&intermediate_hdr + intermediate_hdr_offset	hdr_quantity	once or twice per OUTMSG	putting 1, 2 or 3 B of header information into the hdr_buffer, depending on whether the CPS-Packet header is split or not
extractv	hdr_buffer.hec	hec	5 b	once per OUTMSG	extracting the HEC for verifying
computeCRC	first 19 b of hdr_buffer	computed_crc	order = 5	once per OUTMSG	computing HEC for comparison
extractv	intermediate_hdr.li	li	6 b	once per OUTMSG	extracting Length Indicator in order to determine the number of Bytes in the CPS-Packet payload
assemblep	&INMSG.DU.INFO.payload + read_offset	&OUTMSG.DU.INFO.cps-info + write_offset	info_quantity	once per OUTMSG	mapping parts of the INMSG into the OUTMSG
extractv	intermediate_hdr.uui	uui	5 b	once per OUTMSG	extracting User-to-User-Indication
extractv	intermediate_hdr.cid	OUTMSG.SIDEINFO.cps-cid	1 B	once per OUTMSG	mapping the cps-cid from hdr_buffer into the OUTMSG
assemblev	uui	OUTMSG.SIDEINFO.cps-uui	5 b	once per OUTMSG	mapping the User-to-User-Indication into the OUTMSG
extractv	&INMSG.DU.INFO.payload + read_offset	pad_test	1 B	once per OUTMSG	extracting one Byte from INMSG in order to recognize padding
send	OUTMSG			once per OUTMSG	

Table 36: Data functions AAL2 CPS Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
compare	parity	p	=		verification of the received parity bit
send_notification					Layer Management has to be notified if certain errors occur
update	sequencenumber				the sequence number has to be computed in order to verify the received sequence number
compare	sequencenumber	sn	=		verification of the received sequence number
compare	osf	"47"	=		the value 47 indicates that the current INMSG does not contain a beginning part of a CPS-Packet
compare	computed_crc	hec	=		verification of the HEC
update	read_offset				offset for reading from INMSG
update	write_offset				offset for writing into OUTMSG
update	info_quantity				quantity of data to be extracted from INMSG
update	hdr_quantity				quantity of data to be extracted from INMSG and put into the intermediate header
update	intermediate_hdr_offset				offset for writing to intermediate_hdr
update	expct				quantity of data expected at the beginning of the next INMSG.DU.INFO.payload to complete an overlapping CPS-Packet payload
compare	expct	osf	=		validity check
compare	OUTMSG.DU.INFO.cps-info. LENGTH	Max_SDU_Deliver_Length	$\geq$		check if the data to be sent has a vaild length
compare	uui	"27"	$\leq$		to determine the receiver of the OUTMSG
compare	uui	"30"	=		to determine the receiver of the OUTMSG
compare	uui	"31"	=		to determine the receiver of the OUTMSG
compare	pad_test	"0"	=		If the byte pointed to equals zero, it belongs to the padding

Table 37: Data control functions AAL2 CPS Rx

Function	*Argument*	*Note*
none		

Table 38: Configuration functions AAL5 CPCS Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
assemblep	&INMSG.DU.INFO.id	&OUTMSG.DU.INFO.id	length(INMSG.DU.INFO.id)	once per INMSG	
pad				once per INMSG	Any coding is allowed for the padding. The padding aligns the length of the whole OUTMSG.DU to an integral multiple of 48 B
extractv	INMSG.SIDEINFO.cpcs-uu	OUTMSG.DU.TRAILER.cpcs-uu	1 B	once per INMSG	directly mapped from INMSG to OUTMSG
assemblev	"00000000"	OUTMSG.DU.TRAILER.cpi	1 B	once per INMSG	values other than "00000000" are for further study (I.363.5, 9.2.1.2 d))
computeLength	INMSG.DU.INFO.id	OUTMSG.DU.TRAILER.length	2 B	once per INMSG	OUTMSG.DU.TRAILER.length contains the binary coded length (in Bytes) of the INMSG.DU.INFO.id
computeCRC	OUTMSG.DU.INFO.id & OUTMSG.DU.TRAILER.cpcs-uu & OUTMSG.DU.TRAILER.cpi & OUTMSG.DU.TRAILER.length	OUTMSG.DU.TRAILER.crc	order = 32	once per INMSG	the maximum size of the data the CRC has to be calculated of is 65564 B
assemblev	"0"	OUTMSG.SIDEINFO.m	1 b	once per INMSG	OUTMSG.SIDEINFO.m is always "0" in MM
extractv	INMSG.SIDEINFO.cpcs-lp	OUTMSG.SIDEINFO.sar-lp	1 b	once per INMSG	directly mapped from INMSG to OUTMSG
extractv	INMSG.SIDEINFO.cpcs-ci	OUTMSG.SIDEINFO.sar-ci	1 b	once per INMSG	directly mapped from INMSG to OUTMSG
send	OUTMSG			once per INMSG	

Table 39: Data functions AAL5 CPCS Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
none					

Table 40: Data control functions AAL5 CPCS Tx

Function	*Argument*	*Note*
set	Max_SDU_Deliver_Length	is set by the Management Plane and can take on any value between 1 an 65535
set	Max_Corrupted_SDU_Deliver_Length	OPTIONAL (CDD). Max_Corrupted_SDU_Deliver_Length > Max_SDU_Deliver_Length. Set by the Management Plane

Table 41: Configuration functions AAL5 CPCS Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
assemblep	&INMSG.DU.INFO.id	&cpcs-pdu + cpcs-pdu_offset	48 B	once per INMSG	append INMSG.DU.INFO.id to cpcs-pdu
extractv	INMSG.SIDEINFO.sar-lp	sar-lp	1 b	once per INMSG	to be checked
extractv	INMSG.SIDEINFO.m	m	1 b	once per INMSG	to be checked
discard	cpcs-pdu				needed in some error situations
assemblev	cpcs-pdu.trailer.cpi	rs.val_a	1 B		OPTION(CDD), mapping part of the cpcs-pdu into the rs
assemblev	cpcs-pdu.trailer.length	rs.val_b	2 B		OPTION(CDD), mapping part of the cpcs-pdu into the rs
assemblev	cpcs-pdu.trailer.crc	rs.val_c	4 B		OPTION(CDD), mapping part of the cpcs-pdu into the rs
assemblep	&cpcs-pdu.payload_and_padding	&OUTMSG.DU.INFO.id	length(cpcs-pdu. payload_and_padding)	once per OUTMSG	mapping cpcs-pdu. payload_and_padding into OUTMSG
assemblev	rcv_LP	OUTMSG.SIDEINFO.cpcs-lp	1 b	once per OUTMSG	mapped after having been updated
extractv	INMSG.SIDEINFO.sar-ci	OUTMSG.SIDEINFO.cpcs-ci	1 b	once per OUTMSG	directly mapped
extractv	cpcs-pdu.trailer.cpcs-uu	OUTMSG.SIDEINFO.cpcs-uu	1 B	once per OUTMSG	directly mapped
assemblev	rs	OUTMSG.SIDEINFO.rs	8 B		OPTION(CDD), including the reception status in the OUTMSG
computeCRC	cpcs-pdu.payload_and_padding & cpcs-pdu.trailer.cpcs-uu & cpcs-pdu.trailer.cpi & cpcs-pdu. trailer.length	computed_crc	order = 32	once per OUTMSG	in order to check the received CRC
extractv	cpcs-pdu.trailer.crc	extracted_crc	4 B	once per OUTMSG	to be compared with the computed CRC
extractv	cpcs-pdu.trailer.cpi	cpi	1 B	once per OUTMSG	to be checked
extractv	cpcs-pdu.trailer.length	length	2 B	once per OUTMSG	to be checked
send	OUTMSG			once per OUTMSG	

Table 42: Data functions AAL5 CPCS Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
update	rcv_LP				is to be included inOUTMSG
compare	m	"0"			m indicates whether the received part is the last part of the cpcs-pdu
compare	Max_Corrupted_SDU_Deliver_Length + 8	length(cpcs-pdu)	$\leq$		OPTION(CDD), data too long?
update	rs.flags				OPTION(CDD), the flags indicate which errors occured
update	timer				OPTION: RAS timer
check	timer				OPTION: RAS timer
compare	Max_SDU_Deliver_Length + 7	length(cpcs-pdu)	$<$		when CCD not enabled
compare	crc	computed_crc	$=$		verify the CRC
compare	cpi	"00000000"	$=$		verify the CPI
compare	length	"0"	$=$		user abort? (Streaming Mode)
compare	"0"	length(cpcs-pdu) - length - 8	$\leq$		length valid?
compare	length(cpcs-pdu) - length - 8	"47"	$\leq$		length valid?
compare	Max_SDU_Deliver_Length	length	$\leq$		length valid?
check	rs.flags				OPTION(CCD)

Table 43: Data control functions AAL5 CPCS Rx

Function	*Argument*	*Note*
none		

Table 44: Configuration functions AAL5 SAR Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
extractv	INMSG.SIDEINFO.m	m	1 b	once per INMSG	m=0 indicates that the current INMSG.DU.INFO.id is the last part of data spanning more than one INMSG (only needed in Streaming Mode)
extractv	INMSG.SIDEINFO.sar-lp	OUTMSG.SIDEINFO.slp	1 b	once per OUTMSG	Directly mapped from INMSG to OUTMSG. Could be stored as an intermediate resut, because the same value is needed for every OUTMSG
extractv	INMSG.SIDEINFO.sar-ci	OUTMSG.SIDEINFO.ci	1 b	once per OUTMSG	Directly mapped from INMSG to OUTMSG. Could be stored as an intermediate resut, because the same value is needed for every OUTMSG
assemblep	&INMSG.DU.INFO.id + read_pointer	&OUTMSG.DU.INFO.info	48 B	once per OUTMSG	mapping of 48 B of INMSG.DU.INFO.id to OUTMSG.DU.INFO.info
send	OUTMSG				

Table 45: Data functions AAL5 SAR Tx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
split	INMSG.DU.INFO.id				into fragments of 48 B
LoopControl					
compare	m	"0"	=		m=0 indicates that the current INMSG.DU.INFO.id is the last part of data spanning more than one INMSG (only needed in Streaming Mode)
update	OUTMSG.SIDEINFO.auu				AUU=1 indicates the last part
update	read_pointer				indicates the right position in INMSG

Table 46: Data control functions AAL5 SAR Tx

Function	*Argument*	*Note*
none		

Table 47: Configuration functions AAL5 SAR Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
get	INMSG			once per INMSG	
allocate_mem					
extractv	INMSG.SIDEINFO.auu	auu	1 b	once per INMSG	extracting auu for determining the value of OUTMSG.SIDEINFO.m
extractv	INMSG.SIDEINFO.rlp	OUTMSG.SIDEINFO.sar-lp	1 b	once per INMSG	directly mapped
extractv	INMSG.SIDEINFO.ci	OUTMSG.SIDEINFO.sar-ci	1 b	once per INMSG	directly mapped
assemblep	&INMSG.DU.INFO.info	&OUTMSG.DU.INFO.id	48 B	once per INMSG	directly mapped
send	OUTMSG			once per INMSG	

Table 48: Data functions AAL5 SAR Rx

Function	*Argument*	*Argument*	*Argument/Operator*	*Frequency*	*Note*
compare	auu	"1"	=		last part?
update	OUTMSG.SIDEINFO.m				depending on the value of auu

Table 49: Data control functions AAL5 SAR Rx

C Requirements

The following pages contain the requirements as identified for PPP MUX, PPP MC, AAL 2 and AAL 5.
The abbreviation *i.d.*, which can be found in several tables, means *implementation dependent*.
[x] means *size of x*.

PARAMETERS	range of values	size of parameter	set by	note	reference
MRU	$\leq$ 65535	2 B	config		RFC 1661, p. 41
ACFC	false / true	i.d.	config	OPTIONAL (ACFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 50
PFC	false / true	i.d.	config	OPTIONAL (PFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 48
default_PID		2 B	config	always negotiated, but use is not mandatory	RFC 3153, p. 6
MAX_SF_LEN	$\leq$ min(MRU-2, 16383)	$\leq$ 14 b	user		RFC 3153, p. 4
max_INFO_length	$\leq$ MRU	$\leq$ 14 b	user	OPTIONAL, used for comparison instead of MRU	RFC 3153, p. 5

STATE VARIABLES	range of values	size of variable	note	reference
Last_PID		2 B	("SHOULD")	RFC 3153, p. 4
crc_order	0, 16, 32	i.d.	OPTIONAL (FCS Alternatives)	RFC 1570, p. 6

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG] $\leq$ 2 B + MRU	
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblev	1 b, 1 B, 2 B	1 b, 1 B, 2 B	
assemblep	[location]	$\leq$ MAX_SF_LEN	
extractv	[location]	2 B	
computeCRC	[OUTMSG.DU.HDR & OUTMSG.DU.INFO]	2 B, 4 B	4 B are only needed when OPTION "FCS Alternatives" is used
computeLength	[OUTMSG.DU.INFO.subframe[i].protocolfield & OUTMSG.DU.INFO.subframe[i].info]	6 b, 14 b	
compare	2 B, 17 b	i.d.	
send	[OUTMSG] $\leq$ MRU + 8 B		
check	i.d.	i.d.	
update	i.d.	1 b, 2 B, i.d.	

RESOURCES	count
TIMERS	1 (OPTIONAL)
COUNTERS	0

Table 50: Requirements PPP MUX Tx

PARAMETERS	range of values	size of parameter	set by	note	reference
MRU	$\leq$ 65535	2 B	config		RFC 1661, p. 41
ACFC	false / true	i.d.	config	OPTIONAL (ACFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 50
PFC	false / true	i.d.	config	OPTIONAL (PFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 48
default_PID		2 B	config		RFC 3153, p. 6

STATE VARIABLES	range of values	size of variable	note	reference
Last_rcvd_PID		2 B		RFC 3153, p. 5
crc_order	0, 16, 32	i.d.	OPTIONAL (FCS Alternatives)	RFC 1570, p. 6

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG] $\leq$ 8 B + max(1500 B, MRU)	even if the negotiated MRU is smaller than 1500 B, the receiver must be capable of dealing with INMSG.DU.INFOs that are as big as 1500 B
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblev	2 B	2 B	
assemblep	[location]	$\leq$ max(1500 B, MRU)	
extractv	[location]	1 b, 1 B, 2 B, 4 B	
computeCRC	[INMSG.DU.HDR & INMSG.DU.INFO]	2 B, 4 B	4 B are only needed when OPTION "FCS Alternatives" is used
compare	1 b, 2 B, 4 B, i.d.	i.d.	4 B are only needed when OPTION "FCS Alternatives" is used
discard			
send	[OUTMSG] $\leq$ 2 B + max(1500 B, MRU)		
check	i.d.	i.d.	
update	i.d.	2 B, i.d.	

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 51: Requirements PPP MUX Rx

PARAMETERS	range of values	size of parameter	set by	note	reference
MRU[link]	$\leq$ 65535 B	2 B	config	there might be different MRUs for different links	RFC 1661, p. 41
MRRU	$\leq$ 65535 B	2 B	config		RFC 1990, p. 14
ACFC	false / true	i.d.	config	OPTIONAL (ACFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 50
PFC	false / true	i.d.	config	OPTIONAL (PFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 48
local.OUTMSG.DU.INFO.mphdr.class.WIDTH	2 b, 4 b	i.d.	config	OPTIONAL (Multilink Short Sequence Number Header Format)	RFC 1990, p. 15
local.OUTMSG.DU.INFO.mphdr.zero.WIDTH	0 b, 2 b	i.d.	config	OPTIONAL (Multilink Short Sequence Number Header Format)	RFC 1990, p. 15
local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH	12 b, 3 B	i.d.	config	OPTIONAL (Multilink Short Sequence Number Header Format)	RFC 1990, p. 15

STATE VARIABLES	range of values	size of variable	note	reference
crc_order	0, 16, 32	i.d.	OPTIONAL (FCS Alternatives)	RFC 1570, p. 6
seq_num[class]		3 B		

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG]	
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblev	1 b, 2 b, 1 B, 12 b, 2 B, 3 B, i.d.	1 b, 2 b, 4 b, 1 B, 12 b, 2 B, 3 B	input and output: 12 b only needed when OPTION "Multilink Short Sequence Number Header Format" is used
assemblep	[location]	see note	[INMSG.DU.INFO.fragmentdata] $\leq$ max(1500 B, min(MRU, MRRU)) - 4 B without Multilink Short Sequence Number Header Format Option, [INMSG.DU.INFO.fragmentdata] $\leq$ max(1500 B, min(MRU, MRRU)) - 2 B with Multilink Short Sequence Number Header Format Option
computeCRC	[OUTMSG.DU.HDR & OUTMSG.DU.INFO]	2 B, 4 B	4 B are only needed when OPTION "FCS Alternatives" is used
update	i.d.	1 b, i.d.	
split	i.d.	i.d.	
LoopControl			
increment	12 b, 3 B	12 b, 3 B	
send	[OUTMSG]		

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 52: Requirements PPP MC Tx

PARAMETERS	range of values	size of parameter	set by	note	reference
MRU[link]	$\leq$ 65535 B	2 B	config	there might be different MRUs for different links	RFC 1661, p. 41
MRRU	$\leq$ 65535 B	2 B	config		RFC 1990, p. 14
ACFC	false / true	i.d.	config	OPTIONAL (ACFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 50
PFC	false / true	i.d.	config	OPTIONAL (PFC). Once negotiated, compression is not mandatory to be performed on every frame.	RFC 1661, p. 48
local.OUTMSG.DU.INFO.mphdr.class.WIDTH	2 b, 4 b	i.d.	config	OPTIONAL (Multilink Short Sequence Number Header Format)	RFC 1990, p. 15
local.OUTMSG.DU.INFO.mphdr.zero.WIDTH	0 b, 2 b	i.d.	config	OPTIONAL (Multilink Short Sequence Number Header Format)	RFC 1990, p. 15
local.OUTMSG.DU.INFO.mphdr.seqnum.WIDTH	12 b, 3 B	i.d.	config	OPTIONAL (Multilink Short Sequence Number Header Format)	RFC 1990, p. 15
prefix[class]		i.d.	config	a prefix can be up to 251 B long (limited by the max. size of the LCP option)	RFC 2686, p. 8

STATE VARIABLES	range of values	size of variable	note	reference
crc_order	0, 16, 32	i.d.	OPTIONAL (FCS Alternatives)	
sequencenumber[class]		3 B	is an array!	
minimum[class]		3 B	is an array!	
"list of Bs and Es"	i.d.	i.d.	very implementation dependent how it is kept track of the INMSGs with B and E bits set	RFC 1570, p. 6

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG]	
allocate_mem	i.d.	[INMSG], [OUTMSG], intermediate_memory	might be needed for INMSG, OUTMSG and the intermediate_memory
extractv	[location]	1 b, 2 b, 4 b, 1 B, 12 b, 2 B, 3 B, 4 B	2 b and 4 b are only needed, if Option "Multilink Short Sequence Number Header Format" is used. 4 B are only needed if 4 B FCS Alternative is used.
assemblep	[location]	$\leq$ max(1500 B, MRRU)	
computeCRC	[OUTMSG.DU.HDR & OUTMSG.DU.INFO]	2 B, 4 B	4 B are only needed when OPTION "FCS Alternatives" is used
update	i.d.	3 B, i.d.	
compare	1 b, 2 B, 4 B	i.d.	4 B are only needed when OPTION "FCS Alternatives" is used
check			
discard			
send	[OUTMSG]		

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 53: Requirements PPP MC Rx

PARAMETERS	range of values	size of parameter	set by	note	reference
cps-info.MAXSIZE	45 B, 64 B	i.d.	Mgmt. Plane		ITU-T I.363.2 (11/2000), p. 9

STATE VARIABLES	range of values	size of variable	note	reference
none				

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG] $\leq$ 65568 B + 5 b	
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblev	5 b	5 b	
assemblep	[location]	$\leq$ cps-info.MAXSIZE	
split			
LoopControl			
check			last part?
send	[OUTMSG] $\leq$ cps-info.MAXSIZE + 5 b		

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 54: Requirements AAL2 SSSAR Tx

PARAMETERS	range of values	size of parameter	set by	note	reference
cps-info.MAXSIZE	45 B, 64 B	i.d.	Mgmt. Plane		ITU-T I.363.2 (11/2000), p. 9
Max_SDU_Length	$\leq$ 65568 B	i.d.	Mgmt. Plane		ITU-T I.366.1 (06/98), p. 9

STATE VARIABLES	range of values	size of variable	note	reference
none				

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG] $\leq$ cps-info.MAXSIZE + 5 b	
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblev	5 b	5 b	
assemblep	[location]	$\leq$ cps-info.MAXSIZE	
extractv	[location]	5 b	
discard			
compare	5 b, i.d.	i.d.	
update	i.d.	i.d.	
check			
send_notification			
send	[OUTMSG] $\leq$ Max_SDU_Length + 5 b		

RESOURCES	count
TIMERS	1
COUNTERS	0

Table 55: Requirements AAL2 SSSAR Rx

PARAMETERS	range of values	size of parameter	set by	note	reference
cps-info.MAXSIZE	45 B, 64 B	i.d.	Mgmt. Plane		ITU-T I.363.2 (11/2000), p. 9

STATE VARIABLES	range of values	size of variable	note	reference
permit	false / true	i.d.		

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG] $\leq$ cps-info.MAXSIZE + 13 b	
allocate_mem	i.d.	[INMSG], [cps-packet], [OUTMSG]	might be needed for INMSG, OUTMSG and cps-packet
assemblev	1 b, 5 b, 6 b, 1 B	1 b, 5 b, 6 b, 1 B	
assemblep	[location]	$\leq$ max(47 B, cps-info.MAXSIZE)	
computeLength	$\leq$ cps-info.MAXSIZE	6 b	
computeCRC	[cps-packet.ph.cid & cps-packet.ph.li & cps-packet.ph.uui]	5 b	
computeParity	7 b	1 b	
discard			
pad	i.d.	$\leq$ 47 B	padding has to be '0's
increment			
decrement	6 b	6 b	
update	i.d.	1 b, 6 b, 1 B, i.d.	
check			
get_notification			
send	[OUTMSG] $\leq$ 48 B + 2/3 b		default: 2 b, OPTIONAL: 3 b when ci used

RESOURCES	count
TIMERS	1
COUNTERS	0

Table 56: Requirements AAL2 CPS Tx

PARAMETERS	range of values	size of parameter	set by	note	reference
Max_CPS-SDU_Length	45 B, 64 B	i.d.	Mgmt. Plane		ITU-T I.363.2 (11/2000), p. 22
Max_SDU_Deliver_Length	45 B, 64 B	i.d.	Mgmt. Plane		ITU-T I.363.2 (11/2000), p. 22

STATE VARIABLES	range of values	size of variable	note	reference
sequencenumber	0, 1	1 b		
expct		i.d.		
hdr_quantity		i.d.		

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG]	
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblev	5 b	5 b	
assemblep	[location]	47 B	
extractv	[location]	1 b, 5 b, 6 b, 1 B, 2 B, 3 B	
computeCRC	19 b	5 b	
computeParity	7 b	1 b	
discard			
compare	1 b, 5 b, 6 b, 1 B	i.d.	
update	i.d.	1 b, 6 b, i.d.	
send_notification			
send	[OUTMSG]		

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 57: Requirements AAL2 CPS Rx

PARAMETERS	range of values	size of parameter	set by	note	reference
none					

STATE VARIABLES	range of values	size of variable	note	reference
none				

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		[INMSG] $\leq$ 65536 B + 2/3 b	2 b in Message Mode, 3 b in Streaming Mode
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblev	1 b, 1 B	1 b, 1 B	
assemblep	[location]	$\leq$ 65560 B	
extractv	[location]	1 b, 1 B	
computeLength	$\leq$ 65560 B	2 B	
computeCRC	$\leq$ 65564 B	4 B	
pad		$\leq$ 47 B	Any coding is allowed for the padding. The padding aligns the length of the whole OUTMSG.DU to an integral multiple of 48 B
send	[OUTMSG] $\leq$ 65568 B + 3 b		

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 58: Requirements AAL5 CPCS Tx

PARAMETERS	range of values	size of parameter	set by	note	reference
Max_SDU_Deliver_Length	$\leq$ 65535 B	i.d.	Mgmt. Plane		ITU-T I.363.5 (08/96), p. 30
Max_Corrupted_SDU_Deliver_Length	i.d.	i.d.	Mgmt. Plane	OPTIONAL (Corrupted Data Delivery Option)	ITU-T I.363.5 (08/96), p. 31

STATE VARIABLES	range of values	size of variable	note	reference
none				

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		48 B + 3 b	
allocate_mem	i.d.	[INMSG], cpcs-pdu, rs, [OUTMSG]	might be needed for INMSG, OUTMSG, cpcs-pdu and rs
assemblev	1 b, 1 B, 2 B, 4 B, 8 B		1 B, 2 B, 4 B, 8 B only needed, when Option "Corrupted Data Delivery" is used
assemblep	[location]	$\leq$ Max_SDU_Deliver_Length / Max_Corrupted_SDU_Deliver_Length	depending on whether Option "Corrupted Data Delivery" is used
extractv	1 b, 1 B, 2 B, 4 B	1 b, 1 B, 2 B, 4 B	
computeLength			
computeCRC	[cpcs-pdu.payload_and_padding & cpcs-pdu.trailer.cpcs-uu & cpcs-pdu.trailer.cpi & cpcs-pdu.trailer.length]	4 B	
discard			
compare	1 b, 1 B, 2 B, 4 B	i.d.	
update	i.d.	1 b, i.d.	
check	i.d.	i.d.	
send	[OUTMSG]		

RESOURCES	count
TIMERS	1 (OPTIONAL)
COUNTERS	0

Table 59: Requirements AAL5 CPCS Rx

PARAMETERS	range of values	size of parameter	set by	note	reference
none					

STATE VARIABLES	range of values	size of variable	note	reference
none				

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		$\leq$ 65568 B + 3 b	
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblep	[location]	48 B	
extractv	[location]	1 b	
compare	1 b	i.d.	
update	i.d.	1 b, i.d.	
split			into fragments of 48 B size
LoopControl			
send	48 B + 2/3 b		default: 2 b, OPTIONAL: 3 b when ci used

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 60: Requirements AAL5 SAR Tx

PARAMETERS	range of values	size of parameter	set by	note	reference
none					

STATE VARIABLES	range of values	size of variable	note	reference
none				

Memory requirements for working on the data units: see respective data structures in Appendix A

FUNCTIONS	size of input	size of output	note
get		48 B + 3 b	
allocate_mem	i.d.	[INMSG], [OUTMSG]	might be needed for INMSG and OUTMSG
assemblep	[location]	48 B	
extractv	[location]	1 b	
compare	1 b	i.d.	
update	i.d.	1 b	
send	48 B + 3 b		

RESOURCES	count
TIMERS	0
COUNTERS	0

Table 61: Requirements AAL5 SAR Rx