The Execution Model of APZ/PLEX -An Informal Description

Johan Erikson and Bo Lindell
{johan.erikson, bo.lindell}@idt.mdh.se

Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden

Abstract

Programming Language for EXchanges, PLEX, is a pseudo-parallel and event-driven real-time language developed by *Ericsson*. The language is designed for, and used in, central parts of the AXE telephone switching system. The language has a signal paradigm as its top execution level, and it is event-based in the sense that only events, encoded as signals, can trigger code execution. Due to the fact that a PLEX program file consist of several independent subprograms, in combination with an execution model where new jobs are spawned and put in queues, we also classify the language as pseudo-parallel.

The language PLEX and the AXE system has been the subject of study both in a number of master thesis projects and in several other publications. However, only brief descriptions of the execution model of PLEX have been presented in these works.

The language and its execution model are tightly connected and it is not possible to separate one from the other. This report presents a thorough description of fundamental parts of the language and it also serves as a detailed introduction to the execution model of PLEX.

Contents

1 Introduction **1**

2 The AXE System **1**

2.1 Central- and Regional Processors 2

2.2 The Application Modularity (AM) Concept 4

2.3 Input and Output statements 5

2.4 Load, Reload and Dump 7

3 Programming Language for EXchanges **8**

3.1 The structure of a PLEX program 9

3.2 Records, Files and Pointers 11

3.3 Variables 12

3.4 Data Encapsulation 15

4 The Execution Model **15**

4.1 PLEX structure and OS requirements 16

4.2 Software Units 16

4.3 Function Blocks 18

4.4 Application System 18

5 Program Interwork - Signals **18**

5.1 Direct and buffered signals 20

5.2 Unique and multiple signals 21

5.3 Single and combined signals 22

5.4 Local and Non-local signals 24

5.5 Signals and Priorities 24

5.6 Signals and Data 24

6 Jobs, Signal Buffers and Job Handling **24**

6.1 What is a Job? 25

6.2 Signal Buffers 25

6.3 Job Handling 27

6.4 Execution Time Limits 31

7 Linking Encapsulation **31**

7.1 Addressing a Program Sequence 32

7.1.1 Addressing in DS 35

8 Software Recovery **35**

8.1 Forlopp 37

8.2 System Restart 37

8.3 Forlopp Release or a System Restart? 39

8.4 Variables and Software Recovery 41

A The Signal Description **43**

1 Introduction

The programming language PLEX (Programming Language for EXchanges) is a pseudo-parallel and event-based real-time language developed by *Ericsson* in the 1970's. The language is designed for telephony systems and the dialect studied in this report, PLEX-C, is used in the Central Processor[1] (CP) of the AXE switching system from Ericsson. The language has a signal[2] paradigm as its top execution level, and it is event-based in the sense that only events, encoded as signals, can trigger code execution. The term pseudo-parallel has arisen due to the fact that a PLEX program file consist of independent sub-programs (which will be discussed in Section 5, and Fig. 13), in combination with an execution model (Fig. 21) where new jobs are spawned and put in different queues, called job buffers, for later execution.

The language has been the subject of study in a number of master thesis projects at *Mälardalen University*, [KO00, AGG99, AE00], as well as in a number of research publications, e.g. [MH01, EFGL02]. However, only brief descriptions of the execution model of PLEX have been presented in these works. This is probably due to space limitations and/or the scope of the work in question.

The aim of this report is to give a more thorough description of fundamental parts of the language than the above mentioned works. It also serves as a detailed introduction to the execution model of PLEX. A second aim is to serve as a common basis for future investigations of the language.

Since much of the material in this report is compiled together basically from different forms of internal Ericsson documents, we give these references once and for all at the beginning. If other material is used, these references will be given when used. The references used in this report are [AB99, AB95a, AB98, AB95b, AB02].

2 The AXE System

The AXE telephone exchange system from Ericsson, developed in its earliest version in the beginning of the 1970s, is structured in a modular

[1] The different kind of processors are covered in Section 2.1.

[2] Signals are covered in Section 5.

and hierarchical way. It consists of the two main parts:

APT: The telephony or switching part

APZ: The control part including central and regional processors

which both consist of hardware **and** software. The two main parts are divided into subsystems.

A subsystem is divided in function blocks. Function blocks consist of function units which is either a central software unit or a hardware unit, a regional software unit and a central software unit. The original structure of the system is shown in Fig 1.

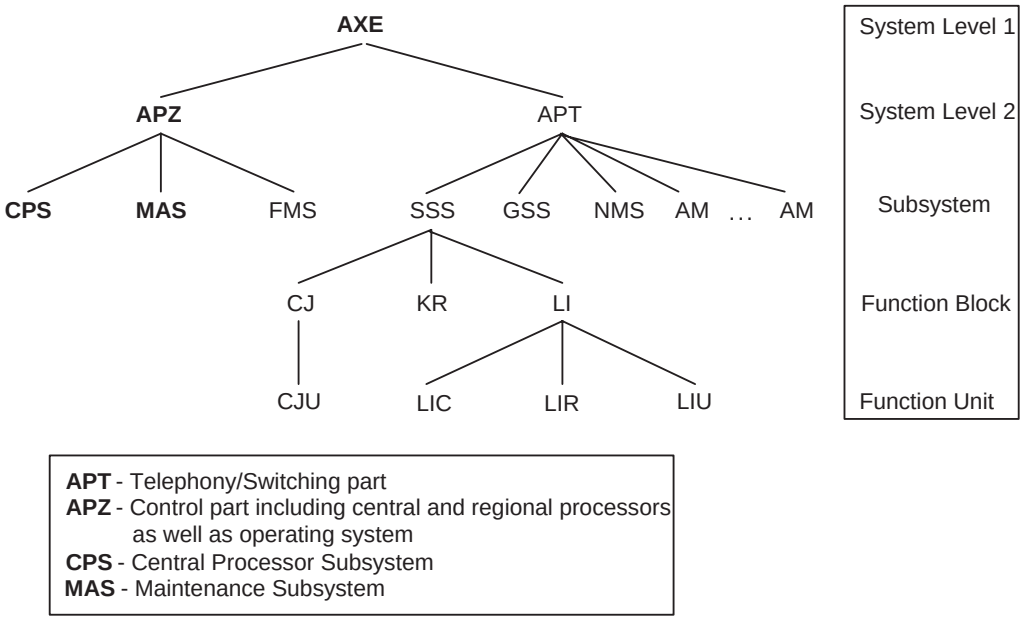


Figure 1: *The (original) hierarchical structure of the AXE system. (The parts that will be of interest in this report is marked with bold text.)*

Somewhere around 1994-95, the concept of *Application Modularity* (AM) was integrated into the system. This will be discussed in Section 2.2

2.1 Central- and Regional Processors

The hardware aspects that is of interest in this report is the distinction between Central- and Regional Processors. This is because different

forms of interwork is performed between different kinds of processors. The distinctions are briefly discussed in this subsection and explained in more detail in Section 5.

Regional Processor (RP): There are several regional processors in an AXE system. The main task of a regional processor is to relieve the central processor by handling small routine jobs like scanning and filtering.

Central Processor (CP): This is the central control unit of the system. All complex and non-trivial decisions are taken in the central processor. This is the place for all forms of non-routine work. The work of the processor can be separated into two specifically distinct parts, namely instruction execution and job administration. Instruction execution means handling of uninterrupted sequences of operations where the work consists of address table look-up and calculations, plausibility checks, storage accesses and data manipulations. The job administration mainly consists of signal handling, signal conversion and signal buffer handling. The execution of instructions is a single-stream work by nature, whereas the job administration to a great extent is a question of prioritized job queues (Section 6) and transfer of signal data.

The CP is always duplicated. The two sides work in parallel, performing exactly the same operations. During normal operation, one CP is executive and the other is stand-by. A continuous check is made to ensure that both processors reach the same result - If they don't, some form of recovery action is performed (Section 8). The CP duplication also enables function changes (installation of new software versions) while the exchange is in an operational mode by first installing new software on the stand-by side and then change the executive and stand-by order between the processors. As a last step, the new software is installed on the former executive (now stand-by) side.

The CPs store all central software and data. The CP memory consists of the register memory and the different stores. Programs are stored in the program store (PS) and data is stored in the data store (DS). The reference store contains information about where

to find the different programs and data, Fig. 2.

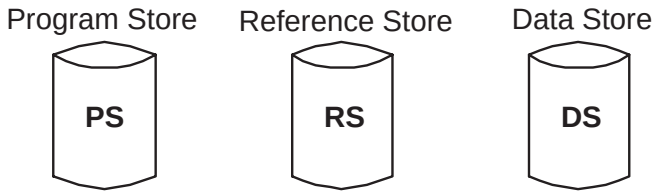


Figure 2: *Stores in the central processor (CP). (The interaction between the different stores are covered in Section 7.)*

2.2 The Application Modularity (AM) Concept

The AXE *Source System* is a number of hardware **and** software resources developed to perform specific functions according to the customer's requirements. It can be thought of as a "basket" containing all the functionality available in the AXE system. Over the years, new source systems has been developed by adding, updating or deleting functions in the original source system. But in the 1980's, the development of the AXE system for different markets (US, UK, Sweden, Asia, etc.) has led to parallel development of the source system since functionality could not easily be ported between different markets.

The solution to this increasing divergence was the *Application Modularity* (AM) concept, which made fast adaption to customer requirements possible. The AM concept specifically targeted the following requirements:

- the ability to freely combine applications in the system,
- quick implementation of requirements, and
- the reuse of existing equipment.

The basic idea is to gather related pieces of software (and hardware) into something called Application Modules (AMs). Different telecom applications, such as ISDN, PSTN (fixed telephony), and PLMN (Public Land Mobile Network), are then constructed by combining the necessary AMs. The idea is described in Fig. 3, where it is also shown that different AMs can be used in more than one application.

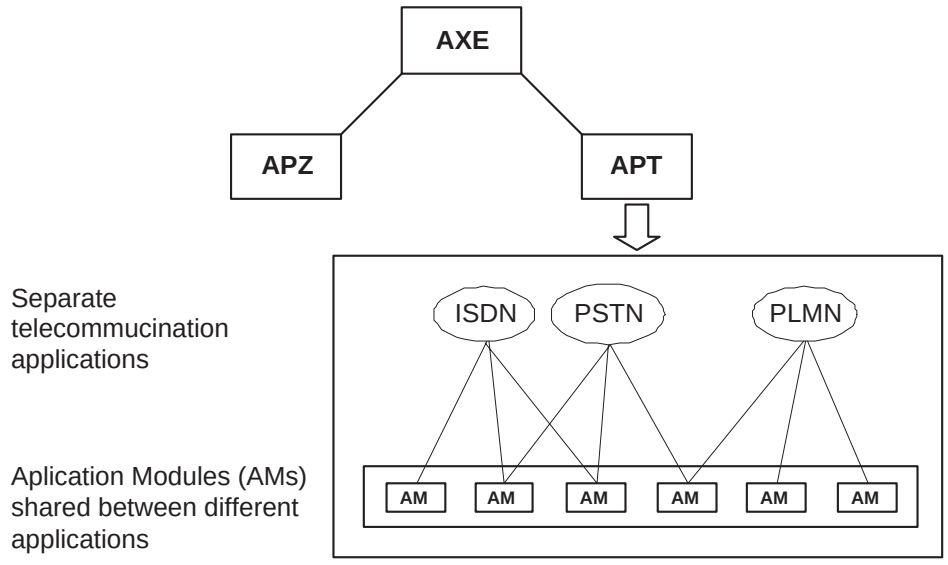


Figure 3: *The AM concept incorporated into the AXE system.*

The introduction of the AM concept ended the problem with parallel development of different source systems. Instead, with AMs as building blocks, the required exchange was constructed by combining the necessary AMs into an exchange with the required functionality (i.e., with the necessary applications).

2.3 Input and Output statements

An AXE exchange needs to communicate with its environment and its operation and maintenance (O&M) staff. Some typical situations could be the following:

- An exchange technician changes subscriber categories, replaces devices or connects new subscribers.
- The exchange informs the O&M staff of important events, e.g., if an RP is blocked due to a fault. In other words, the I/O statements are an important part of the recovery mechanism. (See Section 8.)
- Input/output includes certain routine tasks to, e.g. dumping data on a hard disk.

There is a large number of I/O devices used; alarm and hard copy printers, display units, work stations and PC's, magnetic tape drivers, hard

and flexible disks.

Before communicating with an I/O device, the PLEX program has to seize the device. Likewise, the device has to be released when the communication ends. This guarantees exclusive access to the device. All I/O devices are connected to a support processor (SP), and function blocks that receive or send information via the I/O system are called **user blocks**. Fig. 4 shows the interaction between the I/O system and a user block. When seizing an I/O device, the I/O system assigns a free

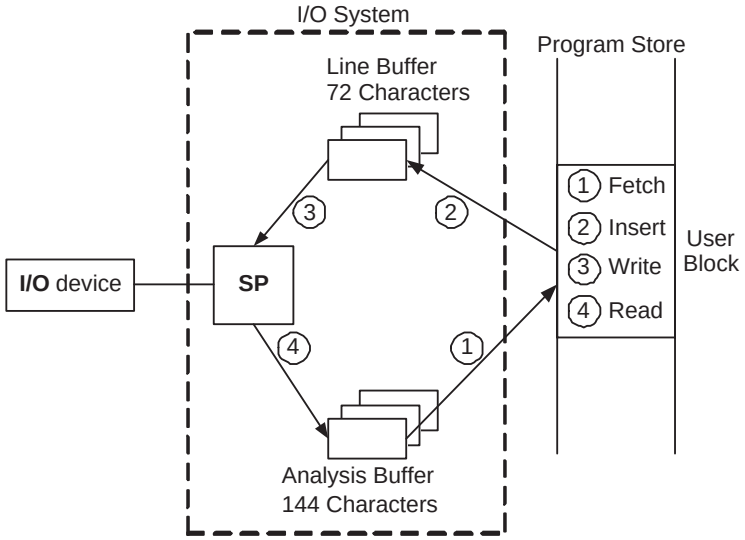


Figure 4: *The I/O system and its communication with the environment.*

line buffer and a free analysis buffer (see Fig. 4) to this device. These buffers temporarily store the I/O text. The analysis buffer handles input from the I/O device, and the line buffer handles output.

The basic (PLEX) statements for transferring information between the buffers and the I/O device, and between the buffers and the user blocks are:

- `FETCH`: transfer information from the analysis buffer to the user block.
- `INSERT`: transfer information from the user block to the line buffer.
- `WRITE`: orders the I/O system to print out the text in the line buffer to an I/O device.
- `READ`: transfer information from the I/O device to the analysis buffer.

Again, see Fig. 4.

Typically, I/O communication starts with the operator entering a command on an I/O device. The command is received by the I/O system and delivered to the software unit where it has been defined by the programmer. A command is received in a program (i.e., a software unit) in the same way as a signal (Section 5) but the command receiving statement must be preceded by the keyword **COMMAND** to indicate that this is a statement used by the I/O system.

2.4 Load, Reload and Dump

An AXE exchange may exist for up to 40 years, which implies certain requirements regarding the operation and maintenance of the software. The terms **Load**, **Reload** and **Dump** are covered in this section since they will be used in this report when we discuss variables (Section 3.3) and software recovery (Section 8).

When all the software blocks have been written and compiled, the programs and data, initial and exchange, are written, *dumped*, to a magnetic tape which is loaded into the exchange. This process is called ***initial loading***. On loading of new blocks, or new revisions of existing blocks, an incremental re-linking occurs, as well as an initialization of data store variable values, if required according to their given *variable properties*[3]. A DCI (Data Conversion Information) is written for each block being loaded to specify the data initialization between the old (if existing) and new blocks. During the *function change process* (Section 2.1) the new block can get its new value from either of the following three ways:

- Get value from *data sector*[4].
- Get value from DCI.
- Get value from existing software.

In the case of system failure where a *system restart*[5] has been performed, software backup copies are *reloaded* into the exchange. When reloaded, some variables will receive reload values from the magnetic tape, whereas other variables will not have values until the program

[3]Variable properties is covered in Section 3.3

[4]The data sector is mentioned in Section 3.1

[5]The system restart process is explained in Section 8

is executed by a *signal*[6]. Whether or not a variable receives a reload value is determined by the variable properties set by the designer. This is covered in Section 3.3.
Reloading means that the contents of DS (i.e., only RELOAD declared variables) are reloaded into the exchange again. If a change has occurred in PS and RS, they will be reloaded as well.

The contents of Program-, Reference- and Data store are regularly saved to a hard disk (or a magnetic tape). This process is called *dump* and enables the reload action described above.

3 Programming Language for EXchanges

Programming Language for EXchanges (PLEX) is designed by Ericsson and used to program telephony systems. It lacks common statements from other programming languages such as `WHILE` loops, negative numeric values and real numbers. These are not needed in a telephony exchange system. The language was designed and developed in its first form in the 1970s and extended in 1983. The version under consideration in this report, PLEX-C, is used in the AXE central processors (see Section 2.1). Other languages used in the AXE system are shown in Fig. 5[7]. The reason for developing a new language for the AXE system was that no other languages under consideration fulfilled Ericsson's requirements.

Some important characteristics of the language are listed below:

- PLEX is an event-based language with a signaling paradigm as the top execution level. Only events can trigger code execution and events are programmed as signals. A typical event is when a subscriber lifts the phone to dial a number.
  The execution model is described in Chapter 4 and signals in Section 5.

- The signals are executed on one of four priority levels (explained in Section 6), which results in very little overhead when a higher

[6] Signals are examined in Section 5

[7] As could be seen in Fig. 5, there is another dialect of PLEX (PLEX-M). However, these dialects are similar, and when we talk about PLEX in this report, we mean the dialect used in the central processors, i.e, the PLEX-C dialect.

level interrupts a lower since each priority level has its own register set.

- Jobs (Section 6.1) at the same level are ”atomic” and can never interrupt each other.

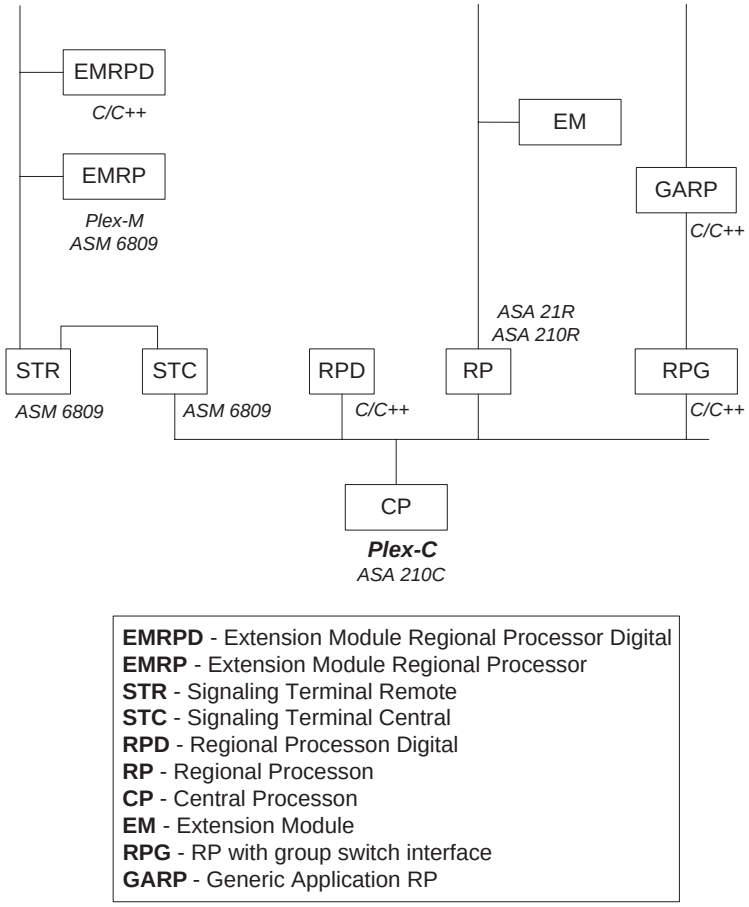


Figure 5: *The different languages used in different parts of the AXE system*

3.1 The structure of a PLEX program

When we talk about a PLEX program, or a PLEX program file, we mean the PLEX file that specifies a function unit (Section 4.3). This document, the *Source Program Information* (SPI), shown in Fig. 6, consists of the following main parts:

- The **Declare** sector, which contains the variable and constant declarations that are used in the program sector. Variables with the property DS, Data Store, (Section 3.3) will exist beyond the execution of subprograms.
- The **Parameter** sector, where specific AXE parameters are placed. These parameters are not local to a block, and permit global access from all parts of the exchange. They can be *changed by customers* since they are placed in an SQL database.
- The **Program** sector contains the executable statements, i.e., the PLEX source code that will run in the exchange. This sector is normally divided in several subprograms (explained in Section 5 and Fig. 13).
- The **Data** sector: Some variables, i.e. Data Store variables, needs to have initial values when the program (i.e., the SPI) is loaded into the exchange[8]. These initial values can be provided in the data sector. Also, the position, i.e. the base address, of stored variables in memory can be allocated in the data sector. This enables a faster function change (briefly described in Section 2.1).
- The **ID** sector is used for internal documentation only.

The SPI is compiled together with the following documents[9]:
- The *Signal Survey*, SS, which is a list of all the different signals that one function unit (i.e., the function unit specified in the SPI) receives and sends. There is one SS per function unit. There is no information about senders and receivers in the SS, this information is added later during loading.
- The *Signal Description*, SD. The function blocks and function units communicate with signals (Section 5). The SD describes the purpose, type and data of *one* signal. SDs are stored in separate signal handling libraries.

[8] The *initial loading* is described in Section 2.4.

[9] The different steps of the compilation process, as well as the PLEX compiler, is described in [AE00]

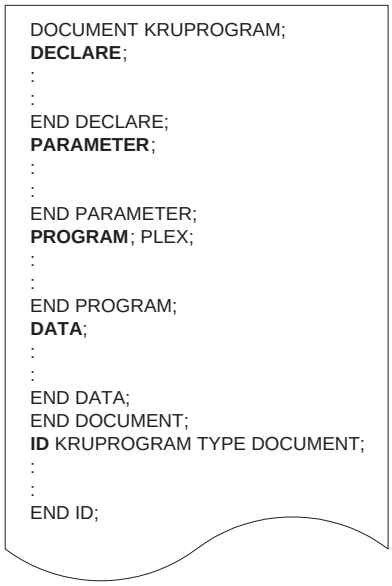


Figure 6: *Structure of the SPI, i.e., a PLEX program file.*

3.2 Records, Files and Pointers

Records collect variables that describe properties of a group of items, for instance, calls or subscribers[10]. Record variables may be stored field, symbol or string variables (Section 3.3). Variables in a record may be indexed or structured, and they are called individual variables. DS (Data Store, described in Section 3.3) variables that are not part of a record, are known as *common* variables.

A **File** is a set of records. One file consist of one or more records, all with the same individual variables.

Pointers address the relevant record in a file. In PLEX, pointers are simply record numbers. The records in a file are numbered, and the value of the pointer is the number of the current record. In other words, pointers in PLEX are **not** similar to pointers in C and can not be manipulated in the same way. Fig. 7 shows an example file with its records and a pointer. The number of records in a file may be fixed or changeable. A fixed size is specified in the Data sector of the SPI (Section 3.1), while alterable file sizes are set by commands (Section 2.3).

[10] A (PLEX) record is similar to a `struct` in C.

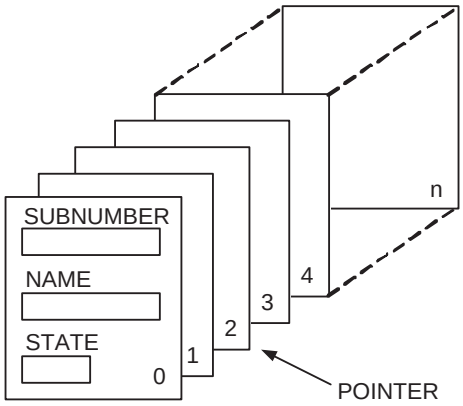


Figure 7: *An example file with n records and a pointer with the current value 2.*

3.3 Variables

Depending on how variables is to be treated at a software error and a following recovery action, the PLEX designer can assign different properties to the variables. This is to be covered in this section.

There are three different data types in PLEX:
- *Field variables* for numeric information. They contain non-negative integers only. (Negative integers are not needed in the AXE system.)
- *Symbol variables* for symbol information, e.g., `IDLE`, `BLOCKED`, `BUSY`, etc.
- *String variables* store text strings.

These data types (variables) can be *stored* or *temporary*.

- The value of a temporary variable exists only in the Register Memory (RM - internal CP registers) and only while its corresponding software is being executed. Variables are by default temporary.
- Stored variables are stored in the Data Store (Fig. 2), loaded into a register in the RM for processing and then written back to the DS. Thus, its value is never lost, even if the program is exited and re-entered later. DS variables are also a natural way to communicate between different *forlopps*[11].

[11] Forlopps are explained in Section 8.1

It is the stored variables that may be assigned the different properties already described. These properties are `DS`, `CLEAR`, `RELOAD`, `DUMP`, `STATIC`, `BUFFER` and `COMMUNICATION BUFFER`. The properties will all be described in this section.

From a storage point of view, the variables can be divided into the following types: Temporary and stored have been described above. The third category is the buffers. Buffer variables[12] are allocated dynamically in an area reserved for dynamic buffers by using an allocate statement. The size of the buffers can be specified static (`COMMUNICATION BUFFER`) or dynamic `BUFFER`. The fixed size is specified in the Declare sector (Section 3.1) while the dynamic size can be set in the Program sector. The dynamic buffers are slower than the static since they must be administered dynamically. These categories are pictured in Fig. 8 together with its properties.

Under normal circumstances, the exchange starts the (application) software and it never stops. After serious errors, however, the APZ (i.e., the operating system part) stops the program execution and restarts the software. The following properties describe the variable behavior at start or restart:

- `CLEAR` - ”Clearing at start/restart”
  Field variables are set to zero; symbol variables to the first value in their declaration list.

- `RELOAD` - Loading at ”restart with reload”
  The variable value is reloaded from tape/hard disk to ensure that the values before and after the ”restart with reload” are the same.

- `DUMP` - ”Dumping at restart”.
  This property is used for testing and tracing purposes.

- `STATIC` - When a software unit in an operating exchange is to be updated, a *function change* takes place. Remember from Section 2.1 that the CP is always duplicated. This means that new software can be installed while the exchange is running. A `STATIC` declared variable means that the variable value is not updated with a new software version.

[12] Buffer variables are similar to the `array` structure in C.

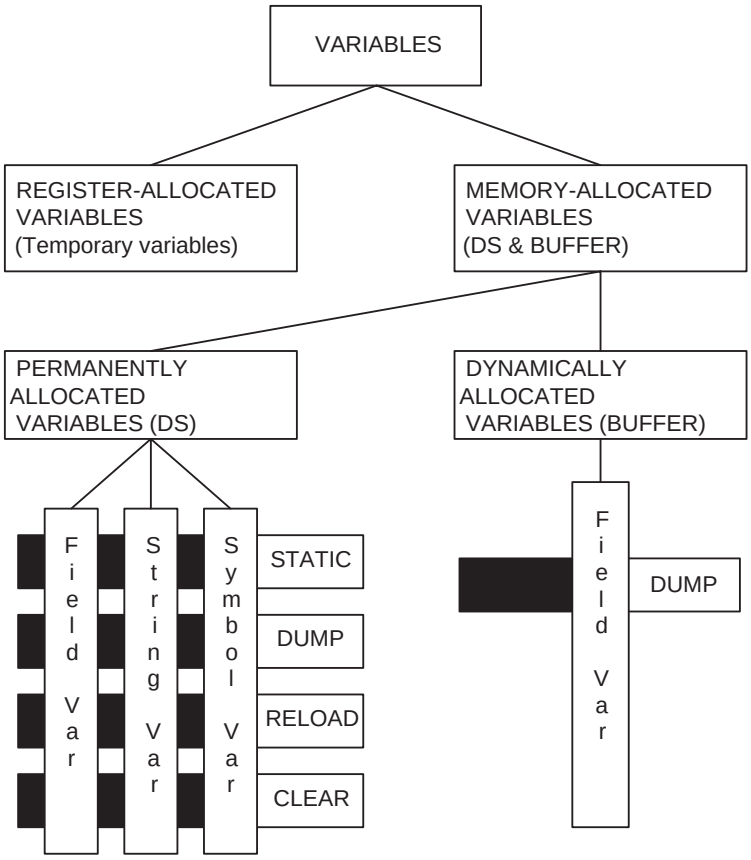


Figure 8: *Variables and properties (from a storage point of view).*

Not all combinations of the variable properties are possible (i.e., legal). Fig. 9 contains a table listing all valid combinations of variables and properties.

	Field Variable	Symbol Variable	String Variable
DS DS DUMP DS STATIC	Yes		
DS RELOAD DS RELOAD DUMP DS RELOAD STATIC	Yes		
DS CLEAR DS CLEAR DUMP	Yes		No
BUFFER BUFFER DUMP	Yes(1)	No	
Temporary	Yes		No

(1) Except for one- and two-dimensional arrays

Figure 9: *Permitted combinations of variable properties and variable types.*

3.4 Data Encapsulation

All variables and constants declared in the *Declare* sector of the SPI, see Section 3.1, have their scope inside the software unit specified. All subprograms (Section 5) of that SPI can access these variables and constants. Subprograms not part of that function unit cannot access these variables and constants.

4 The Execution Model

A brief discussion of the execution model has already been given in Section 3 and we continue and deepen the discussion in this section. We first briefly discuss PLEX structure, operating system requirements, function blocks and application system before we look deeper at program interwork (i.e, *signals*), Section 5, and job buffers, Section 6, both central concepts in the PLEX/APZ environment.

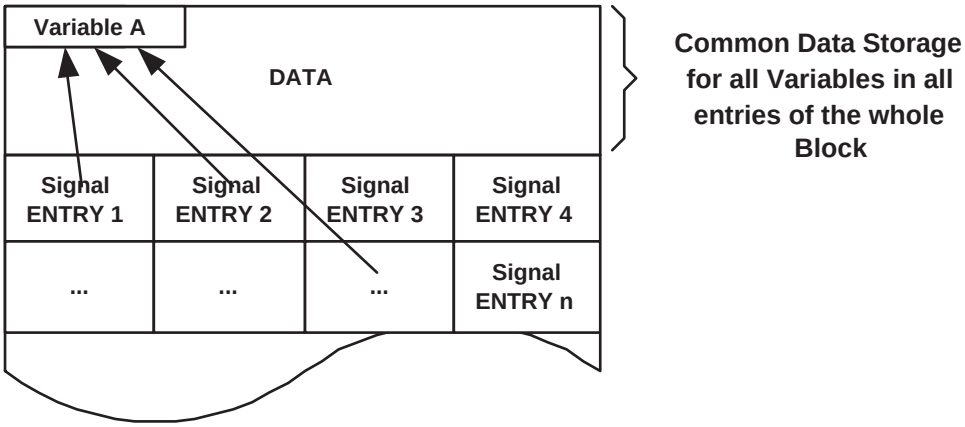


Figure 10: *The structure of a software unit (block). The possibility of several sub-programs accessing the same data within the block is shown. All sub-programs (signal entries) can access all DS variables inside the same block (except for individuals that are DS variables inside a record). This conveys a DS variable can be used as a communication channel between all sub-programs inside the same software unit.*

4.1 PLEX structure and OS requirements

PLEX is an asynchronous concurrent event based real-time language and, as stated in Section 3, it has a signaling paradigm as the top execution level which means that only events can trigger code execution and these events are programmed as signals. Signals will be further explored in Section 5. The main task of an operating system that is to run PLEX, is to buffer incoming signals and start their execution in the right signal entry statement.

4.2 Software Units

In large software systems, such as a telecommunication system, there is a need to group code into modules, for example, to control a certain hardware, or to implement in software add-on functionality. A Software Unit is a quantity of PLEX code for the different jobs[13] needed for such a module, called a function. A Unit can not access data in another unit, i.e, a unit has data encapsulation (see Section 3.4).

[13] Jobs are covered in Section 6.1.

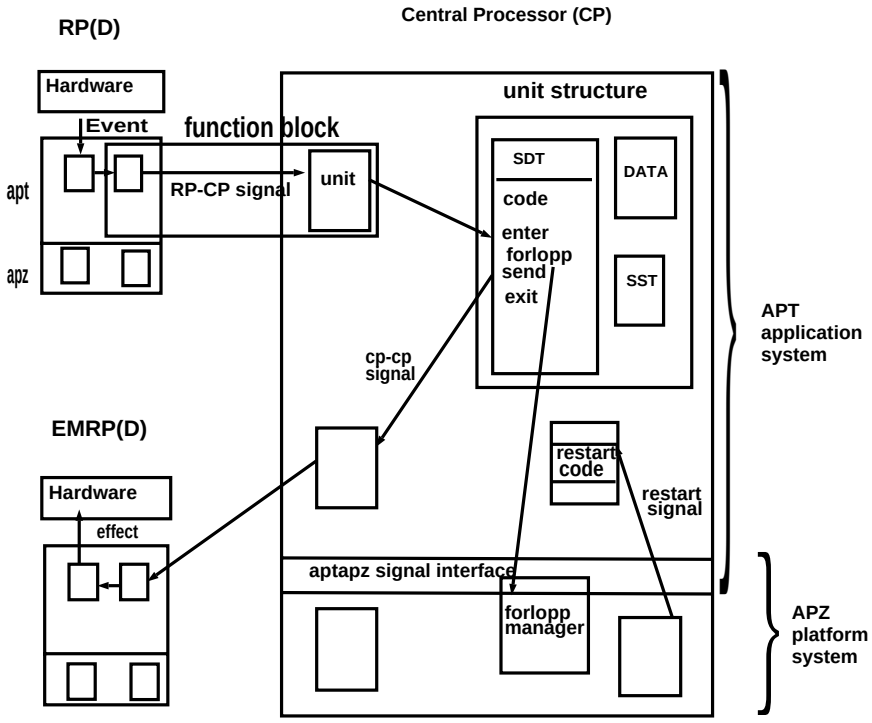


Figure 11: *APT Application system*.

4.3 Function Blocks

A function block is a software unit by itself or a software unit in the CP with the associated software unit in the EMRP or RP and possibly associated hardware needed to implement a function.

If we relate the function blocks to the AM concept, described in Section 2.2, it should be pointed out that an AM is not a PLEX language construct. From a PLEX language point of view, each AM and the common resources can be seen as a collection of blocks. Signals between AMs and to/from the common resources are gathered into standard interfaces.

4.4 Application System

An application system is a group of function blocks that interwork together to form a complete application, such as the control of a certain telephone exchange, see Fig. 11. All the signals and units of the part of the application system hosted on a certain processor take part in a ”linking” process. (For units written in PLEX-C, the host is the CP.) The linking process resolves that signals sent from a certain unit are directed to the right entry point in the right unit.

5 Program Interwork - Signals

A signal is an externally defined language element in PLEX for the interwork between software units. A signal can be described as a message within one or between two software units or as an asynchronous (one way) function call, i.e., it is signals that perform the communication between different function units. Signals can be classified in numerous ways (Section 5.1, 5.2, 5.3 and 5.4) but the **main distinction** is between *direct* and *buffered* signals (Section 5.1). A direct signal is similar to a jump from one function unit or program to another, whereas a buffered signal is more like a `fork`[14] system call **except** that the execution continues in the ”*parent* process” whereas the ”*child* process” is put in the

[14] `fork` is a nonANSI C function that ”*copies the current process and begins executing it concurrently*”, [KP96]. The execution will then continue in this newly created ”child-process”.

job queue (Section 6) for later execution. In this way, after the sending of the buffered signal, the two execution paths are independent parallel threads, unsynchronized with each other. The difference is explained in more detail in Section 5.1, but we already state that buffered signals is the ”norm” and that the classification referred to only applies to CP-CP signals. CP-RP and RP-CP signals are **always** buffered.

As shown in Fig. 12, signals are sent between software executing on the different processor types described in Section 2.1.

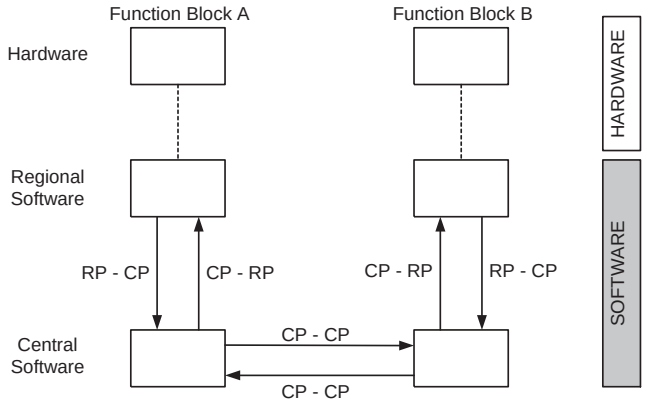


Figure 12: *The different types of software signals.*

Most signals could be seen as a jump from a signal-sending statement in one program to a signal-receiving statement in another program (even if buffered signals first go through a buffer). This implies that the code in a PLEX program unit[15] never executes from the beginning to the end (i.e., from the beginning of the program file to the end of the program file), but from a signal receiving statement (e.g., `ENTER`), to either a direct signal-sending statement (e.g., `SEND`) or an `EXIT` statement. In PLEX, a ***subprogram*** is the code sequence from `ENTER` to `EXIT`. It is possible to leave a subprogram with an `EXIT` without a previous signal sending statement, but it is also possible to send several buffered signals before an `EXIT` statement. Fig. 13 illustrates a general program divided into subprograms. Note that since programs written in PLEX do not normally execute from start to end, or in any order, it can not be assumed that the program in Fig. 13 receives SIGNAL1 before or

[15] A PLEX program unit = a PLEX source code file

after SIGNAL3, or SIGNAL4 before or after SIGNAL6. This can result in unpredictable values of stored variables.

```
PROGRAM; PLEX;
   ENTER SIGNAL1;
   ....
   SEND BUFFERED SIGNAL2;      a subprogram
   ....
   EXIT;

   ENTER SIGNAL3;
   ....                        a subprogram
   SEND DIRECT SIGNAL4;

   CUSELESS = 0;

   ENTER SIGNAL5;
   ....
   SEND BUFFERED SIGNAL6;      a subprogram
   ....
   SEND DIRECT SIGNAL7;

   ENTER SIGNAL8;
   ....                        a subprogram
   EXIT;

   ....
   END PROGRAM;
```

Figure 13: *A PLEX program file divided in subprograms. Note that the assignment* `CUSELESS = 0;` *will never be executed since it is placed between an exit and an enter statement. (See also Fig. 6 where a complete program file is described.)*

Since the exchange handles several calls simultaneously while the CP can only execute one program at a time, the CP must queue the signals somewhere. This is done in job buffers, a job table or in time queues and this will be explored in Section 6.

As was said earlier there are different parameters that describe the signal properties of a CP-CP signal. Three groups classify these properties and each signal has one property from each group. Each group is described below and all possible combinations is shown in Fig. 17.

5.1 Direct and buffered signals

As was stated in Section 5, the main distinction between (CP-CP) signals is whether they are direct or buffered. Buffered signals start **a new**

job, whereas direct signals **continue** the current job. (Jobs are covered in Section 6.1). That is, they are handled differently in the execution model.

Direct signals reach the receiving block immediately, they could be seen as direct jumps to another unit. By using direct signals, other signals have no possibility of coming-in-between, i.e., the programmer *retains control* over the execution. However, direct signals are normally only allowed to be used in very time-critical program sequences, such as call set-up routines.

With buffered signals, it is not predictable when the signal reaches the receiving block. Direct and buffered signals are illustrated in Fig. 14.

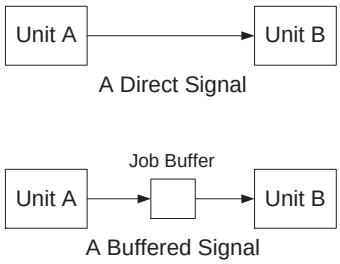


Figure 14: *Direct and buffered signals.*

5.2 Unique and multiple signals

This distinction concerns the number of receivers of the signal. A unique signal can only be received in one particular block, while a multiple signal can go to any block as shown in Fig. 15. However, it is not possible to send a multiple signal to more than one block simultaneously which means that a multiple signal **does not** perform multicast[16]. But even if a multiple signal can go to any of the receiving blocks specified in the *Signal Survey*[17], the signal sending statement must always contain one (**and only one**) receiver of the multiple signal.

[16]Multicast: Send once - received by all

[17]The Signal Survey is described in Section 3.1

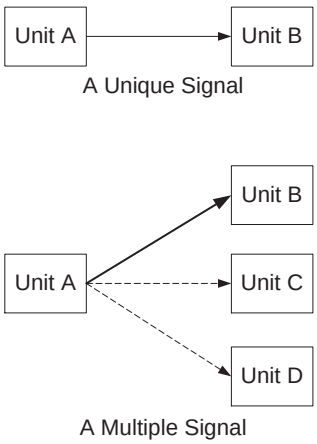


Figure 15: *Unique and multiple signals.*

5.3 Single and combined signals

The third distinction concerns whether the sending block expects an answer. Combined signals demand an immediate answer, while single signals do not require such feedback. For this reason, combined signals can **never** be buffered (as shown in Fig. 17). Instead, they behave like direct jumps from one unit to another. When the execution in the other unit (the receiver of the signal) finishes, execution jumps back to the originating unit. Combined signals are always direct signals, which means that execution continues without interrupt and all other signals have to wait. Fig. 16 illustrates these kind of signals.

When discussing the sending and receiving of combined signals, one will also mention *forward* and *backward* signals. A communication between two parts[18] is always initiated by one of the parts. The initiating part is sending the forward signal whereas the part that replies to the call is sending the backward signal. This is pictured in Fig. 18.

[18]Which, in our target domain, is the sending and receiving of signals between *function blocks*.

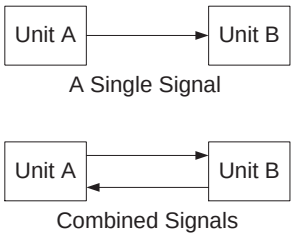


Figure 16: *Single and combined signals.*

Signal Type		Direct	Buffered
Single	unique	X	X
	multiple	X	X
Combined	unique	X	
	multiple	X	

Figure 17: *Possible properties for CP-CP signals. X indicates a legal/possible combination, shaded with Grey indicates an illegal alternative.* **NOTE**: *A combined* **backward** *signal can not be multiple since this signal is an answer (i.e., an acknowledgment) to a "caller" and must therefore return to the "caller" and nobody else.*

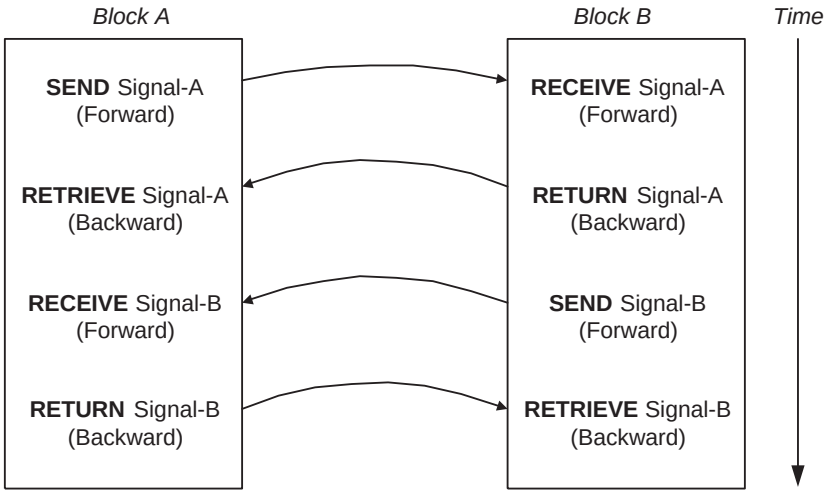


Figure 18: *Forward and Backward signals.*

5.4 Local and Non-local signals

In the beginning of Section 5, we said that signals are used ”for the interwork between software units”. But signals can also be used for the interwork between *different parts of the same software unit*. These signals are called *local signals*, since they are local to the software unit they belong to. I.e., the recipient resides in the same software unit. (Consequently, all other signals are called *non-local* signals.

The behavior of a local signal is similar to that of a GOTO statement since they result in direct jumps to the recipient. (And in that sense, they can be regarded as direct signals.)

Whether a signal is local or not, is specified in the Signal Description (which was briefly explained in Section 3.1, and covered in more detail in Appendix A). The distinction between local and non-local signals is of importance in, for instance a semantic framework for PLEX.

5.5 Signals and Priorities

Every signal that is sent in the system is assigned a priority level, A - D. The priority level is of importance when the signal is to be buffered (Section 6), and it tells the ”importance” of the source code that is triggered to execution by the signal. The priority of each signal is specified in the corresponding Signal Description.

5.6 Signals and Data

Signal Data are variable values sent with a signal[19]. The data may consist of field variables, symbol variables, pointers, numerals, string objects, buffer variables and field expressions. For single and combined signals, it is possible to send 25 signal data. The data is loaded to the register memory in the central processor (see Section 2.1) if the signal is direct, **or** to the job buffer if the signal is to be buffered.

6 Jobs, Signal Buffers and Job Handling

In the following sub-sections, we will discuss the definition of a *job* (Section 6.1), the different ways of delaying/buffering a signal (Section 6.2)

[19]This is similar to a *call by value* function call.

and, finally, how jobs are handled at runtime (Section 6.3).

6.1 What is a Job?

A job is a continuous sequence of statements executed in the processor. A job begins with an `ENTER` statement for a **buffered** signal and ends with an `EXIT` statement.

Between the `ENTER` and the `EXIT` statement, several buffered signals (or no signals at all) may be sent. A job is not limited to one CP software unit, several units and blocks can take part in a job.

A job does **always** have a single entry point but it *may* have multiple exit points.

In Section 5.5 we discussed the priority of a signal. In the following subsections, we will instead talk about the priority of a job. This make sense since it is more natural to look at whole jobs when discussing execution of PLEX code, than it is to look at a single[20] signal. The reason is that a job includes the actual PLEX code that is triggered to execution by the signal, as well as the signal itself.

6.2 Signal Buffers

Some jobs in the AXE system are not time-critical and can wait to be executed, while others need to be executed immediately. The first case holds for administrative jobs and the second case for jobs related to traffic handling (i.e., telephone calls[21]) and CP faults.

Buffered signals (which could be read as ”the start of a new job”) may be delayed using one of the following methods:

- Job Buffer: delays a signal until all ”older” jobs have been processed
- Job Table: sends signals at short periodic intervals
- Time Queue: delays signals by relative or absolute time

We will look further to these different ways of delaying a signal.

[20] By single signals, we do **not** mean single signals as described in Section 5.3.

[21] A normal load on the system is 200 telephone calls that is to be handled every second. These jobs are all time critical and have the same priority, but the performance would not be acceptable with a ”*first-come-first-served*” approach. A solutions is to use buffered signals as a ”time sharing” mechanism.

Job Buffers: Job buffers are queues with a **FIFO-semantics**[22]. There are four buffers for CP-CP and RP-CP signals and one for CP-RP signals; *Job Buffer A*, *Job Buffer B*, *Job Buffer C* and *Job Buffer D*, all for CP-CP and RP-CP signals, where Job Buffer A has the highest priority. *Job Buffer R* is the buffer for CP-RP signals.

The buffers carry the following type of tasks:
Job Buffer A - *urgent tasks of the operating system*; preferential jobs, e.g., errors in traffic equipment.
Job Buffer B - telephone traffic.
Job Buffer C - I/O communication. The command statement described in Section 2.3 is handled at this level.
Job Buffer D - APZ routine self-tests.
Job Buffer R - CP-RP signals queue in JBR, a buffer for signals sent from the CP to a RP.

The Job Table: The job table contains jobs executed at short periodic intervals, for instance, incrementing clocks for time supervision. The job table has higher priority than any of the job buffers. Since the possible execution time after a job table signal is very short, this signal only initiates a program sequence in the receiving block, which inserts a buffered signal in one of the job buffers. The buffered signal initiates the ”real” work in the program which from an application point of view, has the priority of the buffer it is inserted in.

Time Queues: Time queues delay periodic and other jobs at longer intervals than the job table. There is one absolute time queue and three relative ones. The absolute time queue stores the absolute time for signal execution (month, day, hour and minute). Every minute, the time queue compares this value with the system calendar. When there is a match, the signal is moved to one of the four job buffers. The three relative queues have a counter for each job. Every 100 ms, 1 second and 1 minute, respectively, the time queue receives a periodic signal from the job table and decrements the counter. If a counter reaches the value zero, the corresponding signal is forwarded to one of the job buffers. I.e., a signal that

[22] First In First Out

is fetched from a time queue is almost never executed at once[23]. Execution of the signal is performed when the operating system fetches it from the job buffer it was inserted in.

Fig. 19 shows how a software unit sends a delayed (and multiple) signal. The signal is first placed in a time queue and after that in a job buffer. After it is taken from the job buffer, the execution is started in the receiving unit.

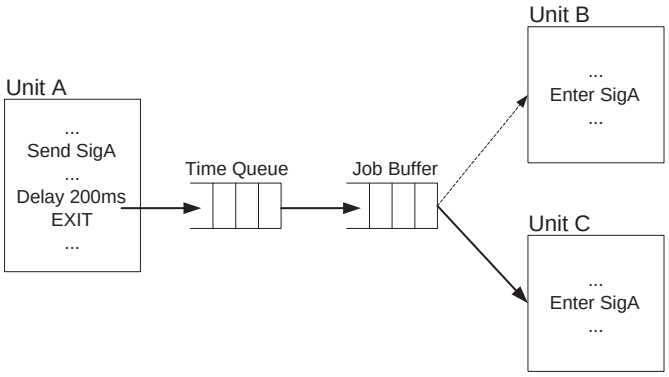


Figure 19: *Sending of a delayed (and multiple) signal. The signal is sent from Unit A and received in Unit C but, as could be seen in the figure, it is possible to receive the signal in Unit B as well* ***if*** *Unit B is specified as the receiver by the PLEX designer.*

6.3 Job Handling

The priorities at runtime correspond to the priorities among the job buffers (Section 6.2), as will be shown below.

As already stated, Section 6.2, depending on their purpose and time requirements, jobs are assigned to certain priority levels - five different levels exist. But the important thing, when dicussing job priorities, is how different priority levels can interrupt each other and, as could be seen in the following discussion, we could view the five different priority levels as only three if we take the possibility for one job to preempt another into consideration.

[23]The only exception is when the receiving job buffer (and every job buffer with higher priority) is empty.

Tasks initiated by a periodic Job Table signal use the traffic-handling level 1 (THL 1), JBA signals use traffic-handling level 2 (THL 2), JBB use traffic-handling level 3 (THL 3), JBC use base level 1 (BAL 1) and JBD use base level 2 (BAL 2), see Fig. 20.

The Job Table has a higher priority than all the job buffers. JBA has a higher priority than JBB, and so forth. The jobs in the job buffers are executed in order of priority - JBA is emptied before JBB, and so on. Data used in interrupted jobs stay in the processor register memory, and THL, BAL 1 and BAL 2 jobs have their own processor registers. That means **all** THL jobs share the same register buffers. Hence, no job at one sub level of THL can interrupt a job at another sub level of THL, since they share the same set of registers and the temporary variables would be destroyed otherwise.

I.e., jobs from the job table, JBA and JBB have to wait for each other, but all three can interrupt job from JBC and JBD. As BAL 1 and BAL 2 have different register memories, JBC can interrupt JBD.

JBA - urgent tasks of the operating system: preferential traffic
JBB - all other telephone traffic
JBC - input/output to operator and I/O devices
JBD - APZ routine self-test
JBR - signals from Central Processor to Regional Processor
THL - traffic-handling level
BAL - base level

Figure 20: *Job buffers and runtime priorities in the AXE system.*

In some cases, however, it may be necessary to prevent the system from interrupting an important task. For example, an operation and maintenance (O&M, Section 2.3) routine at C-level (BAL 1) is writing to variables that are also accessed by traffic-handling routines at B-level (THL 3). In this situation, it is best to inhibit the interrupt function as

long as the writing at C-level is in progress. The interrupt function is inhibited by the `DISABLE INTERRUPT` statement and activated by the `ENABLE INTERRUPT` statement.

We conclude this subsection with an exampel. Fig. 21 illustrates the execution of several jobs. In the figure, the execution starts in block 1 with the first job, proceeds in block 2 with the second job and finally ends in block 1 with the execution of the last job. Fig. 22 gives a closer look of the link (into job buffers) and execute process.

If a new job enters an empty job buffer, the buffer sends an interrupt signal for that priority level. If the ongoing job has a lower priority level, that job is interrupted. However, a job can not interrupt a job on the same (or higher) priority level.

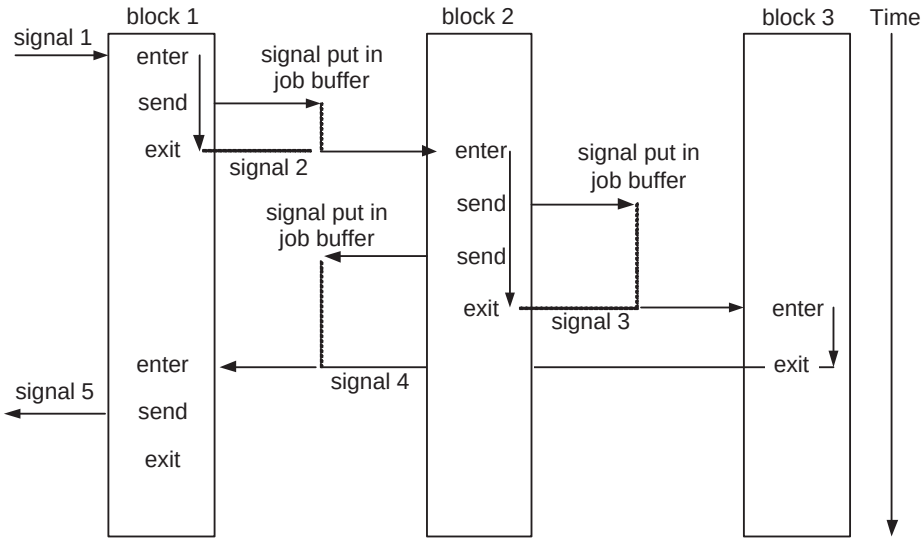


Figure 21: *The execution model - Four jobs are executed. The process of transfering a buffered signal from the sending block to the receiving, via a job buffer, is shown in Fig. 22. NOTE the "parallel" architecture that could become real parallel execution.*

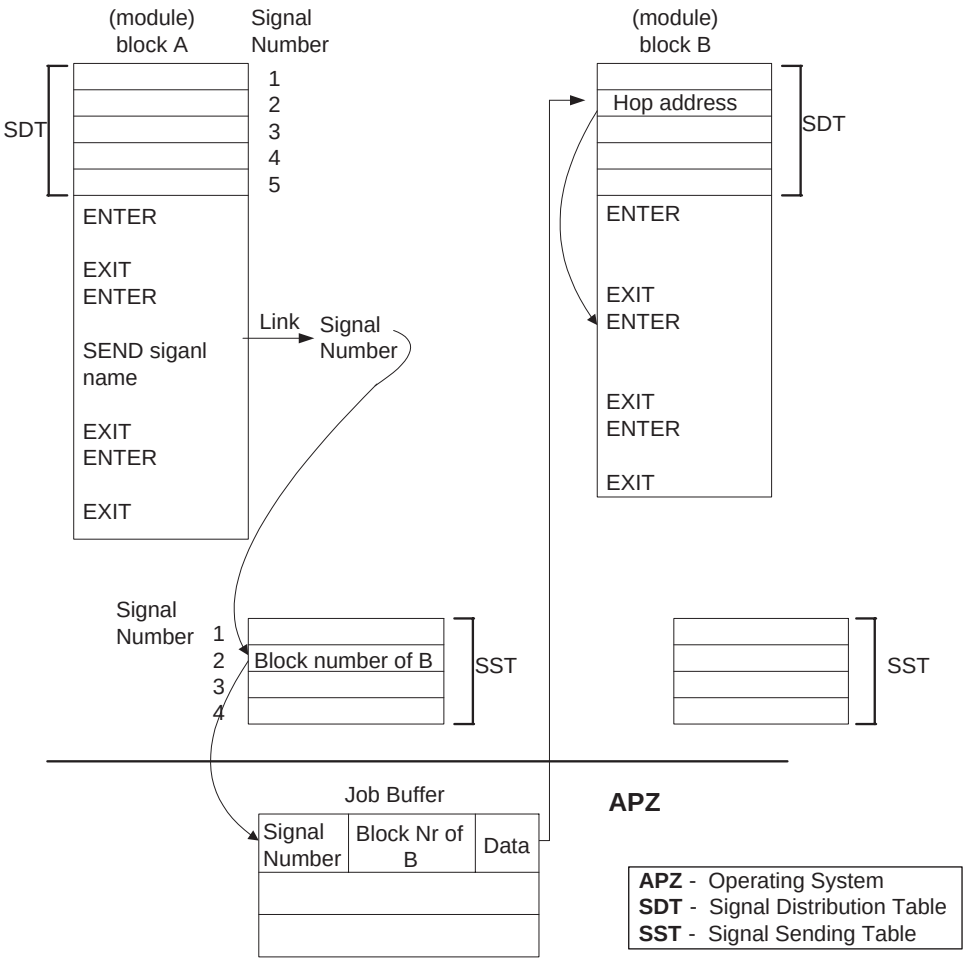


Figure 22: *Linking and execution for a buffered signal in APZ. See also Fig. 21.* **NOTE**: *The procedure is the same for direct signals except that they* ***not*** *are inserted in a Job Buffer.*

6.4 Execution Time Limits

As stated in Section 4.1, PLEX is a real-time language. This means that a system programmed in PLEX is a real-time system[24]. When talking about execution times limits, one always refer to the execution time of a job. There are limits for the execution time, but this is not measured in absolute times. Instead, there are programmer guidelines that specify how many lines of code that may be placed in a software unit (or units) for one job.

7 Linking Encapsulation

All blocks used in the system are compiled separately and it is also possible to ”load“ them separately, even at run-time. This process is called a Function Change and it was described in Section 2.1. When doing a Function Change, the Signal-Sending Table (SST) and the Global-Signal Distribution Table (GSDT) has to be updated. The update has to be done because all signal sendings has to look in the SST and the GSDT to find which signal to invoke.

When updating the tables, by the Rationalized Software Production (RSP) functionality, the (new) introduced signal is given a unique number, the Global Signal Number (GSN). This number is stored in the GSDT as well as in the SST of the Function Unit (block) using this ”new“ signal. The GSDT also stores Block Number Receiving (BN-R), (the unique number of the block receiving the signal) and the Local Signal Number (LSN) which is the position holding the local relative address of the entry point of the signal entry.

The Signal Distribution Table (SDT) is *not* updated, as the SDT holds the relative address to the signal entries inside the Function Unit. SDT is set with a local number in the object step (during compilation).

SDT: Contains the relative entry address, set during compilation, of the specific program sequences where signals are received.

SST: Contains the global signal number (GSN) of signals to invoke

[24] And, as shown by Arnström et. al, the AXE system is classified as a *soft real-time* system [AGG99].

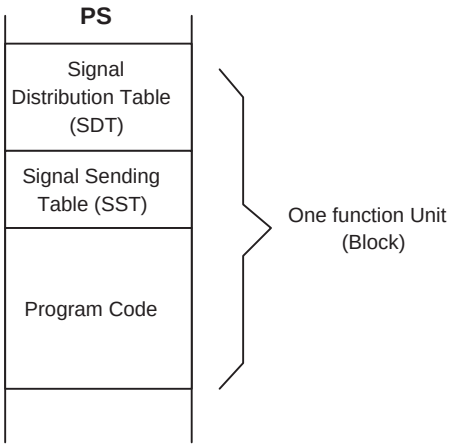


Figure 23: *PS, showing SDT, SST and Program Code of one function unit.*

from function unit using ”this“ SST, created in the object step and changed by the RSP.

GSDT: Contains the global signal number (GSN), the Block Number Receiving (BN-R) and the Local Signal Number (LSN).

In DS, values are stored for all variables.

In PS, the programs for all blocks are stored together with the Signal-Sending Tables (SST), the Signal Distribution Table (SDT) and the Global-Signal Distribution Table (GSDT), see Fig. 23

RS is used for addressing DS and PS, and contain the Program Start Address (PSA) and Base Start Address (BSA), see Fig. 24.

7.1 Addressing a Program Sequence

Fig. 25 shows ”unit A” sending a signal to ”unit B”; the global signal number (GSN) is found in the Signal-Sending Table (SST) of ”unit A”. The GSN is used to find the Block Number Receive (BN-R) and the Local Signal Number (LSN) in ”unit B” (”unit A” doesn't know it is ”unit B” that holds the signal entry for the signal sent from ”unit A”). The BN-R is used to obtain the Program Start Address (PSA) in the Register Store (RS). The PSA is an absolute address in the Program Store (PS), and by knowing the LSN and PSA, and also using the Signal Distribution Table

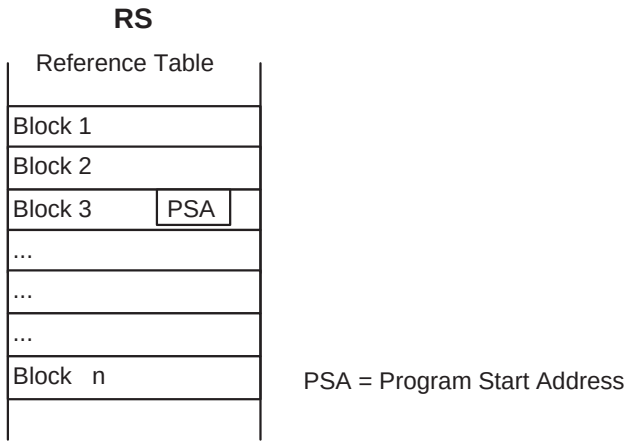


Figure 24: *RS, showing the Reference Table.*

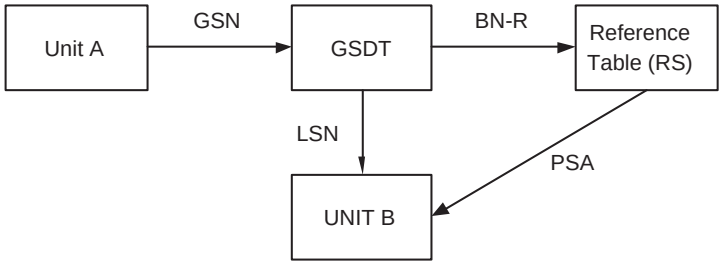


Figure 25: *The information flow in determining the signal entry when sending a signal.*

(SDT) of ”unit B” the entry point of the program code can be determined in ”unit B”. See Fig. 26.

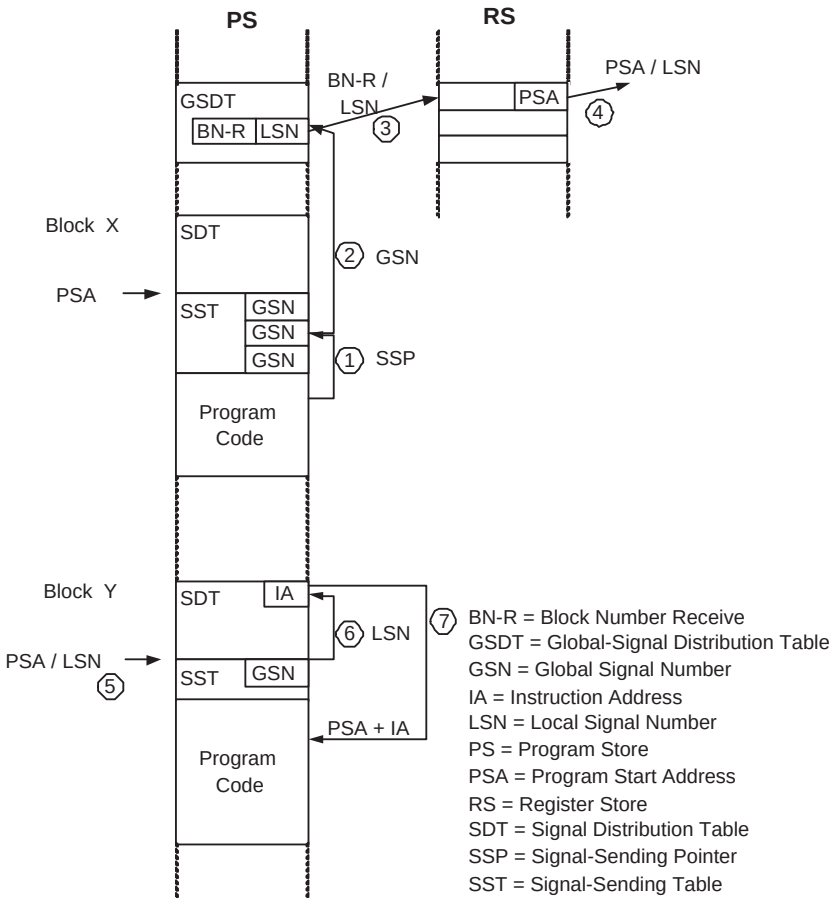


Figure 26: *The consecutive order of handling a signal sending.*

7.1.1 Addressing in DS

RS actually consists of two parts: the Reference Table (RT) and the Base Address Table (BAT). In the RT there is one PSA and Base Start Address (BSA) for each block, and the BSA points to the starting point of BAT, see Fig. 27[25].

RT: is part of Register Store and hold the Program Start Address and Base Start Address.

BAT: holds the address of the variables in DS. For each block a variable is given a number from 1 and upwards. This number is called the *Base Address Number* (BAN). To get the address of a variable in DS, the BSA + BAN will give the position holding the address in DS.

BSA: holds the address of current start point of BAT.

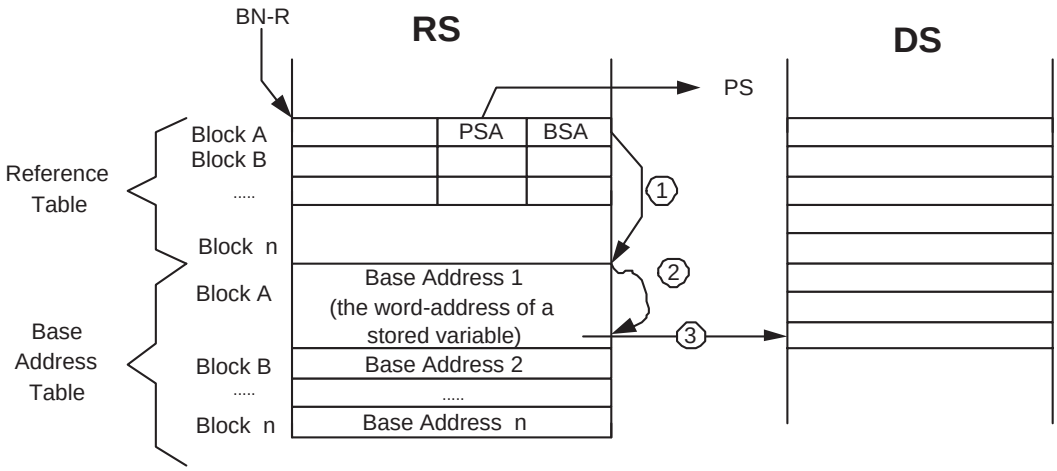
8 Software Recovery

After the initial loading (Section 2.4), the exchange is supposed to run smoothly during its lifetime. This is also the normal situation for the system. However, errors can't be entirely eliminated and in this section we will study software recovery actions. The goal with the automated recovery action is to minimize the exchange down-time. This is achieved by first trying to release only the dysfunctional forlopp[26] (which normally stretches over parts of several blocks) and leave the rest of the system unaffected. As a last step, if nothing else works, the entire system is restarted.

This section will cover the different steps regarding software recovery actions. In Section 3.3 we stated that variables are treated differently at recovery actions depending on the properties set by the designer. We will end this section with a summary of variable properties and their ”behavior” at recovery actions.

[25] Actually, this is how addressing is performed in some architectures. The addressing principles may differ among the APZ versions

[26] Forlopp will be described in Section 8.1



1 = BSA indicates the starting point of base address table for block A located in reference store. The BSA will give the absolute address.
2 = BAN indicates where the BAT word address for the specific variable is found. BAN is a relative address.
3 = The word address indicates where the value of the specific variable is stored in DS.

Figure 27: *Show how addressing to DS is performed in RS. BSA points to the starting point of BAT*

8.1 Forlopp

The first line of defense for maintaining system availability is the *Forlopp* release. The purpose of a forlopp release is to allow a single process chain, e.g., a call, to be released without adversely affecting any other processes in the system.

Forlopp originates from the Swedish word ”förlopp” meaning ”sequence of related events”. In the contents of AXE, a typical forlopp will result in a ”path through the system” which generally will be represented by a chain of linked software resources, such as records. In AXE, the word forlopp can be used to denote both the ”sequence of related events” and the resulting ”path through the system”. The forlopp mechanism is implemented in the Maintenance Subsystem, MAS, Fig. 1. Examples of forlopps are an ordinary telephone call or a command. Some concepts associated with forlopps:

- A forlopp identity (FID), stored in a special register, is assigned to each process (a call or forlopp). All parts participating in the same forlopp have the same forlopp identity.
- The forlopp manager (FM) stores information concerning the different forlopps.
- When a software error is detected, the FM sends *release signals* to the blocks involved according to the information stored in FM. A forlopp release is hereby performed.
- At a forlopp release, a software error dump is performed, which means that the contents of the records participating in the current forlopp are dumped[27].

To summarize, a detected software fault may result in a forlopp release, provided that the function block in which the fault occurred is *forlopp-adapted* and the forlopp function is active.

8.2 System Restart

The *system restart* has been the traditional recovery action taken by the APZ (Section 2) when it detects a software fault. The system restart

[27] Section 2.4 describes what a dump is.

affects the entire system and not only the forlopp in which the fault occurred. The purpose of a system restart is to restore the system to a predefined state.

During restart, *restart signals* are sent to each block, so that during successive restart phases, blocks perform actions to complete the initialization or restoration to a consistent value of their data store variables.

The system restart procedure could be initiated manually, by a COMMAND (Section 2.3), or automatically. A manual system restart clears error situations, for instance the disconnection of a hanging device. An automatic system restart is detected by programs, microprograms and supervisory circuits. At a system restart, the job table, the job buffers and the time queues (Section 6) are cleared.

There are three levels of system restart activities:

- Small system restart, which does not affect calls in speech position and semi-permanent connections. Other calls are disconnected. This is a minimal system restart.
- Large system restart in which all calls are disconnected. Semi-permanent connections are not affected.
- Reload and large system restart in which a reload is performed first to ensure that RELOAD-marked variables contain correct values. This is then followed by a large system restart. Semi-permanent connections are disconnected and automatically reestablished.

The reason to have different types of system restarts is to disturb traffic handling as little as possible during the restart phase.

With the occurrence of the first fault in a normal block that leads to a system restart, the system tries to repair itself without disturbing the traffic too much - A small system restart is initiated. If another serious fault occurs within a predefined time interval, a large system restart will be initiated. In the event of the occurrence of a third serious fault within another predefined time interval, a reload and a large system restart will take place. This represents the system's most extreme error-recovery action. The described phases is pictured in Fig. 28.

Finally, it is sometimes unnecessary to immediately initiate an automatic system restart. The system restart could be delayed or inhibited. This is done by calling the *selective restart* function.

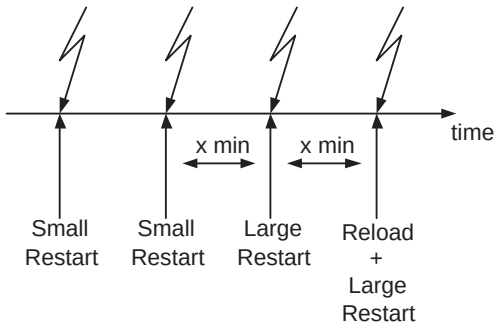


Figure 28: *Different types of system restart.*

8.3 Forlopp Release or a System Restart?

In Section 8.1 we described the concept of forlopps as a way to recover from a software error without affecting more than the faulty forlopp. Then, in the following Section, 8.2, we described the system restart and the different levels of restart and when they apply. This section explains when the system restart action takes over from the forlopp release mechanism.

As we said in Section 8.1, a forlopp release is always a first choice if an error has been detected. The system restart ”function” applies when and if:

- The forlopp release fails to recover the system (i.e., the faulty forlopp), or
- The faulty process has not been forlopp-adapted, or
- The number of faults have been to high according to a predetermined limit.

The last case is checked against an *intensity counter*. This counter keeps tracks of the quantity of software faults. The counter is stepped each time a fault is detected leading to a delayed system restart or a forlopp release. When the counter reaches the predetermined limit, a system restart is initiated. The counter is then reset and starts again from zero. Fig. 29 shows the intensity counter and Fig. 30 shows the different levels of software recovery.

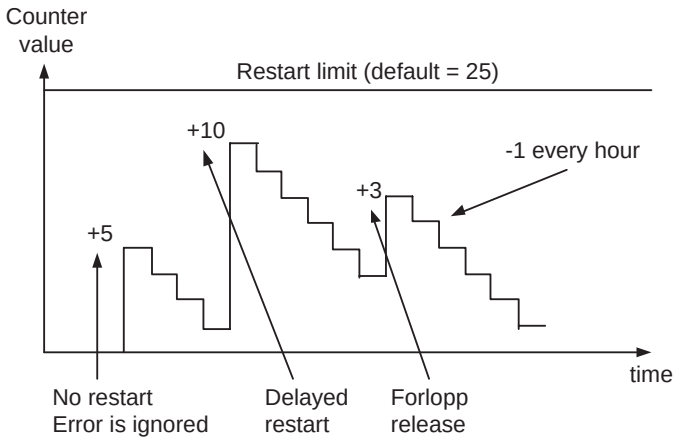


Figure 29: *The intensity counter.*

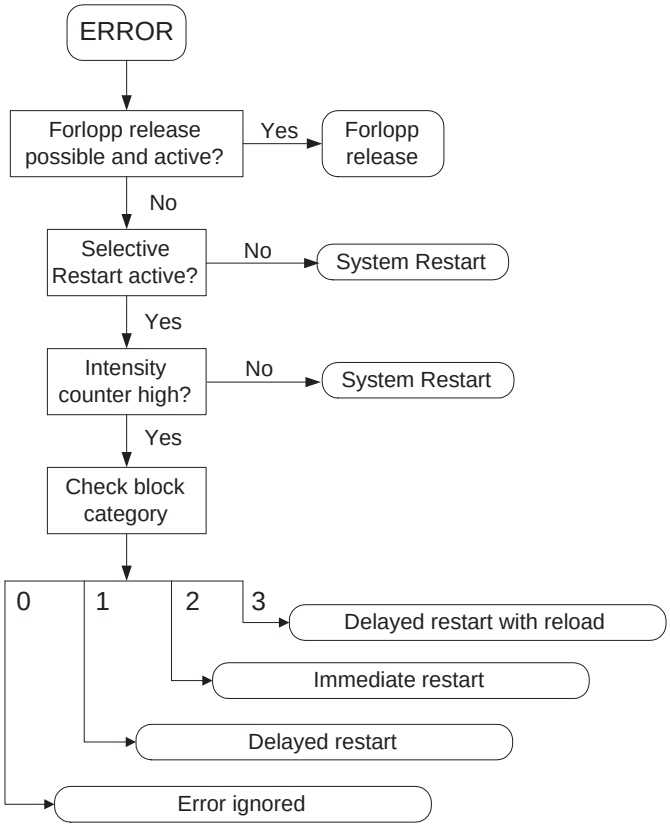


Figure 30: *Different levels of recovery after a detected software error.*

8.4 Variables and Software Recovery

As said in Section 3.3, the variable properties determine how the variables are to be treated (i.e., from a data point of view) in the case of a system restart. Fig. 31 shows the principles of how different types of variables should be treated after a system restart.

<table>
<tr><th></th><th>Start</th><th>Small system restart</th><th>Large system restart</th><th>System restart with reloading</th></tr>
<tr><td>DS
DS DUMP
DS STATIC</td><td>Cannot be trusted</td><td colspan="2">Cannot be trusted
Exception: when the variable value is checked in system restart routine</td><td>Cannot be trusted</td></tr>
<tr><td>DS RELOAD
DS RELOAD DUMP
DS RELOAD STATIC</td><td colspan="3">Can be trusted</td><td>Can be trusted</td></tr>
<tr><td>DS CLEAR
DS CLEAR DUMP</td><td colspan="4">Can be trusted</td></tr>
</table>

Figure 31: *Data security of different start/restart types.*

Acknowledgements

This report is published within the research co-operation between *Ericsson AB* and *Mälardalen Real-Time Research Center*. The work has been founded by ***Ericsson AB***, ***Mälardalen Real-Time Research Center*** and ***the KK-foundation***.

The authors would like to ***Janet Wennersten*** at Ericsson AB as well as professor ***Björn Lisper*** at Mälardalen University for many helpful discussions.

We would also like to mention ***Per Burman***, ***Anders R. Larsson*** and ***Anders Skelander*** at Ericsson AB, as well as ***Peter Funk*** at Mälardalen University, who all have been eager to keep the research co-operation between Ericsson AB and Mälardalen University ”up and running”.

References

[AB95a] Ericsson Telecom AB. *CPS Principles*, 1995.

[AB95b] Ericsson Telecom AB. *PLEX-C 2*, 1995.

[AB98] Ericsson Telecom AB. *PLEX-C 1*, 1998.

[AB99] Ericsson Telecom AB. *Basic Principles of Forlopp Handling*, 1999.

[AB02] Ericsson Telecom AB. *PLEX-C Language Description*, 2002.

[AE00] J. Axelsson and J. Erikson. SAPP, Theories and Tools for Execution Time Estimation for Soft Real Time (Communication) Systems. Master's thesis, Mälardalen University, 2000.

[AGG99] A. Arnstrom, C. Grosz, and A. Guillemot. GRETA: a tool concept for validation and verification of signal based systems (e.g. written in PLEX). Master's thesis, Mälardalen University, 1999.

[EFGL02] J. Eriksson, P. Funk, J. Gustafsson, and B. Lisper. A tool concept for execution time analysis of legacy systems. In *14th Euromicro Conference on Real-Time Systems, Work-in-Progress*. IEEE, 2002.

[KO00] P. Karlsson and S. Ohlsson. Jämförelse av registerallokerigsstrategier för programmerinsspråket PLEX. Master's thesis, Mälardalen högskola, 2000.

[KP96] Al Kelley and Ira Pohl. *C By Dissection - The Essentials of C Programming*. Addison-Wesley, 1996.

[MH01] A. Marburger and D. Herzberg. E-cares research project: Understanding complex legacy telecommunication systems. In *Fifth European Conference on Software Maintenance and Reengineering*, pages 139 – 147. IEEE, 2001.

A The Signal Description

There are two main documents for signals: The *Signal Survey*, which is a listing of all the signals sent and received in a unit, and the *Signal Description*, which will be studied in this section. These documents are compiled together with the *Source Program Information*[28], SPI.

The Signal Description, SD, is the document that *defines* a signal. The type of the signal, as well as the priority level of the signal is specified in this document. There is *one SD for every signal* and all SD's are stored in special libraries. We will study how the SD "interact" with the SPI during the code generation phase. But as a first attempt to capture the contents of the SD, it could be seen as similar to the `h-file` in `C` that externally defines a function (among other things).
The SD includes the following items:

- **Name of the signal** - Every signal has a name that, perhaps, captures its functionality.
- **Signal number** - For internal documentation.
- **Function** - Used to make comments about functionality.
- **Signal type** - The signal type indicates whether the signal is *Single* or *Combined*. There are three possible type specifications:
  - **Type 1**: Single signal
  - **Type 2**: Combined forward
  - **Type 3**: Combined backward

  If the signal is *Multiple*, this is indicated by adding the keyword **MULTIPLE** to the signal type, like in: `TYPE IS 1 MULTIPLE`
  A *Unique* signal has no indication at this point! (A signal is unique "by default" if nothing else is stated.) If the signal is local, the keyword **LOCAL** is added to the signal type.
- *Possible* **return signal** - For internal documentation.

[28]The Source Program Information is the "source code file", i.e., the document that we normally call a program. See Section 3.1 for further details.

- *Possible* **sending block** - For internal documentation.
- *Possible* **receiving block** - For internal documentation.
- **Buffer level** - The priority level! In Section 6, it is described how every signal is assigned a priority level, and how signals are stored in different *job buffers* (**in case of a buffered signal**). The buffer level states the priority level of the corresponding job and also in which job buffer the signal will be buffered (if it is to be buffered, i.e.).
  NOTE: The buffer level can have the following combinations:
  - **NO BUFFER**: The signal is direct. (A combined signal is **always** direct, see Section 5.3.)
  - **LEVEL A/B/C/D BUFFER**: The signal is buffered and uses the job buffer specified. This combination *overrides* a possible use of the `HURRY` option in the signal sending statement (see below).
  - **LEVEL A/B/C/D**: The signal is buffered and uses the job buffer specified, ***unless*** the keyword `HURRY` is used in the signal sending statement, which indicates that the signal is direct.
- **Signal data** - Specification of the ”arguments” (i.e., the data) that the signal is carrying.
- **ID sector** - For internal documentation.