YASP 1.36 INTERFACE

FUNCTIONAL SPECIFICATION

Document # 941111-0

Pierre G. Richard, Christophe D. Espert

April 2nd, 1997

This is a revised version of the document **"THE SGML PARSER INTERFACE FUNCTIONAL SPECIFICATION / VERSION 1.25 / Document Number 920315-4 / March 15, 1992" by Geoff Bartlett and Pierre Richard, IBM T.J. Watson Research Center, Yorktown Heights, New York**.

Following changes have been made:

- More readable formats (Microsoft Word 6.0a and 7.0)
- Error fixes (we noticed that the descriptions were obsolete vs. the YASPAPI include file)
- Reorganization of chapters
- Simplifications (suppressions, but no additions)

These modifications have been made under the sole judgment of the author. In any case, the reader is advised to keep a copy of the original document as a reference.

The SGML Standard (ISO 8879:1986) assumes that a conforming SGML parser processes a document sequentially, while providing information about the structure of the document to the application. The basic set of information a parser must provide was known as the Element Structure Information Set (ESIS)[1]. The YASP interface has been designed to accomplish this main role, furnishing to the application NOTIFICATIONS of SGML events conforming to the ESIS. However the YASP interface goes beyond this requirement, and can provide more information than that needed to achieve SGML conformance. In particular the YASP interface may be used to generate GROVEs as defined in the DSSSL (ISO/IEC 10179 :1996) standard and useful for the HyTime (ISO/IEC 10744 :1992) standard.

ESIS is a list of limited information that applies to systems that process the data sequentially, but is not adequate for interactive SGML applications such as editors or navigators. Also, ISO 8879 says nothing about portability between different operating systems. A conforming SGML parser need not be portable. To allow YASP to be used widely on different systems and for multiple applications, the interface provides functions to:

1. insure system independence.

   Functions that are system dependent are not directly encoded in YASP. This includes memory management, I/O, and message processing. When such functions need to be performed, YASP asks the host system to do it through a set of API services called "System Services" (a good example of such services is presented in the server.c module). This design is very powerful. YASP Applications have been successfully run on many diverse operating systems, and even on host applications that already had their own interfaces with the operating system. Samples of Application Servers are provided to help application designers in their implementations (the SMP00 application unveils the power of the API).

2. provide supplementary notifications needed to "debug" SGML.

   SGML features as SHORTTAG, OMITTAG, SHORTREF, DATATAG can make it quite difficult to follow the Parser actions. For instance, when designing a DTD, it is especially useful to be able to trace the parsing step by step, showing the side effects of a complex content-model. YASP notifications have been extended far beyond ESIS to allow such traces.

3. help an editor to deal with the SGML information contained in the SGML Markup Declaration and Prologs.

   Information about the document syntax and structure is required for complex applications that let the user modify the document instance. Such an application can ask YASP to provide such information in what is called "Parser Services."

[1] ISO/IEC JTC/SC18/WG8/N931, Annex A, Attachment

Table of Contents

1. Concepts

1.1 What is an SGML Parser?

Electronic text formatting for publishing has been evolving steadily over the years, from an early base of formatters that simply describes the layout and presentation of text (procedural markup). These formatters have been extended to provide generic markup capabilities, such as the IBM Generalized Markup Language (GML). In the last few years, the value and importance of "intent based" generic markup has been recognized, and we now have an International Standard Generalized Markup Language (SGML)[2].

SGML describes the rule governing the syntax of a language for marking up documents, and has a number of advantages over its more informal predecessors. The most important of these is that it allows the structure of a valid document to be defined rigorously. This means that a document can be validated, that is, can be shown to conform to a specific Document Type Definition (DTD). The role of an SGML Parser is to insure the validity of the Document, according to its associated DTD.

1.2 Interface design

An SGML Parser allows an application such as a formatter or an editor to rely on the structure of the document that is to be processed. This means an application can be implemented more easily (and can be made more robust—at any point in the process, the state of the document is known).

To make full use of this considerable advantage, the SGML Parser needs to provide the application with the knowledge of the document structure. To achieve this, the SGML Parser must work with well designed structures, while notifying the application of "markup events" as it validates and processes the source text of a document.

In addition, the SGML Parser must not be dependent on a given Operating System, to accommodate alternative applications running in different environments. For this purpose, the Interface must define a standard, system-independent way for the Parser to request system dependent "services," such as I/O, memory management, and message delivery.

System services and application notifications are the main components of the YASP Interface. The SGML Parser calls the two kinds of associated functions, using the same scheme. From the point of view of the SGML Parser, an "application," such as a formatter or an editor, and the "system" are viewed as the same entity (i.e., "the outside world"), globally named the *Application*.

1.3 Relationship of the SGML Parser and the Application

To take full advantage of the power of the SGML Parser, the Parser and the Application must have an interactive relationship. If the Parser processes a document and produces an intermediate form of the document that the Application processes later, the Application is, in effect, reparsing the document.

The SGML Parser Interface provides a design for an Interactive relationship between the Parser and its Application. The Parser and the Application operate simultaneously, and this permits the Application to act immediately on the information the Parser reports to it. This interactive design, as mentioned above, can be used with alternative Applications that support it — and it can also be used with any Parser (SGML or non-SGML) that supports it.

Some Applications, such as "batch" formatters, do not always require the full interactivity of this interface, but their job is made easier by it. Other Applications, such as intent-based or structured editors, or "WYSIWYG" editor-formatters, do need the "incremental" capability of this interactive interface.

The interactive relationship between the Parser and its Application is accomplishing by making the two operate cooperatively as coroutines. At least in CMS, the essence of a coroutine relationship is a blurring of distinction between *calling* and *returning*, which imply a master/servant relationship. In OS2, Windows 3.x or Windows

[2] ISO 8879 (Oct 1986)

NT, the Parser has been implemented as a Dynamic Link Library (DLL) linked together with the Application. There are two models we can think of for the interface between the Parser and its Application:

1. The Application calls the Parser to find the next “interesting thing” in the document. When it finds it, the Parser returns to the Application, which processes it, and repeats the process until the Parser returns the interesting fact that the document is finished.

2. The Parser parses the Document, and calls the Application whenever it finds something “interesting” that needs system processing. The Application performs the appropriate action, and returns to the Parser to continue.

It makes little difference which of these models you prefer. Neither is quite correct and both are correct, for the sort of cooperative relationship that the YASP Interface establishes. The Parser and its Application transfer control back and forth as equals, each performing the tasks that it was designed to do. With the YASP Interface, the Application has the control first, and it is the Application that first calls the Parser to do parsing required in the primary purpose of the Application. In this sense, the Application is the master, and the Parser is the servant. However, it makes the ongoing communication between the Parser and the Application easier to implement if both are continuously active throughout the job. To facilitate this, the Application initially starts up the Parser main procedure which retains control, and processes the entire document. The Parser returns from this main call only when the document is finished, or when a severe error condition is detected. In the meantime, the Parser repeatedly calls the Application through the Interface entry, to report the results of parsing including passing the contents of the document to be processed (Notifications), or to ask for system services. For the YASP parser, the Application must provide services for I/O, memory management and message delivery, which represent the usual collection of system-dependent services.

There is a fourth, highly system-dependent service. It is Startup. The main entry point of the YASP Parser is designed to minimize the system dependencies resulting from the various call-return conventions. At present, we have more than four different kinds of Applications that have been successfully interfaced with YASP. The worst case required a few lines of Assembler to be written. In all cases, keeping the YASP Parser fully reentrant did not lead to implementation of any system-specific code. C applications can call the main entry point of the Parser directly, assuming they have been compiled with the same compiler, and that they provide a big enough stack. Two STARTUP assembler programs (assembler 8086, and assembler 370) are provided to facilitate the implementation of any other application.

The design of the YASP Main entry point and Interface completely isolates the Parser from the System, maximizing its chances of being very portable to any operating system.

2. Overview of the Interface Structures

Communication between the Parser and the Application is governed by a PARSER/APPLICATION INTERFACE, which is described here. The Parser/Application Interface takes the form of a series of defined structures that are used to pass information and requests back and forth.

2.1 Basic sub-structures

2.1.1 Flags

Flag fields carry classification information that is specific to each particular control block. They fall into two categories:

1. For non Boolean values, that is, only one of several possibilities can be true, the field is defined as a series of enumerated values. There is a known range of valid numbers, and each number has a defined meaning. All these values are always in increasing order.
2. For Boolean values, where a single "field" may simultaneously show the existence/nonexistence or true/false state of several conditions, each "flag" is defined as a numeric value that is a power of 2. This is a more portable way than using C bit-fields, that are machine dependent.

2.1.2 Application Handle (AHL)

An Application might need to create its own system dependent data, in response to a Parser request. The Parser might later need to request from the Application a function that is specifically related to this data. There must be means for the Parser to present back to the Application Application-owned data, even if the Parser does not know exactly what is within the data.

For example, in file management functions, an Application may have to create control blocks in response to a Parser OPEN file request. The Parser will later present to the Application those control blocks while asking to READ the file.

The Application Handle structure (AHL) is a basic data type that allows any system to store and retrieve any kind of data. AHL usually conveys such information as either a token (integer) or a pointer (address). Since the Application creates the AHL, and the Parser merely gives back the same AHL on subsequent requests, the meaning and interpretation of the AHL is entirely defined by the Application. The Parser has no need to know how to interpret the AHL created and used in a particular Application.

For this Interface, AHLs are used only to deal with files[3], either

- to create and store a file identification: "FileId-AHL," or
- to remember how to access a given file: "Access-AHL."

The AHL is described by Figure 1. This is a field that can either be an integer or a pointer to any kind of data.

```
typedef union {          /* to pass any kind of file handle */
   void *p;              /* either pointer */
   int   i;              /* or identifier */
}AHL;
```

Figure 1 AHL substructure. application can choose either to store a token or an address to reference its own data

2.1.3 Address-Length (ADLEN)

The standard means of passing data is via an Address/Length pair, called an ADLEN. Standard ADLENs, as described in Figure 2, are part of other interface structures.

[3] See I/O functions for a definition of "file"

The combination of an Address and a Length can describe an arbitrary array of elements of any length. The field ADLEN.P usually points to the first element of the array, and ADLEN.LEN contains the number of elements in the array. For instance, an ADLEN can describe a character string, ADLEN.P being the pointer to the first character and ADLEN.LEN the length of the string. In a few cases, the Parser/Application interface uses an ADLEN to pass a pointer to another structure, rather than an array. In this case, the field ADLEN.P is sufficient to hold the pointer, and the field ADLEN.LEN is either immaterial, or can be used to hold some other information.

```
typedef struct {           /* to pass any lengthed chunk */
   void * p;               /* address */
   int    len;             /* length */
}ADLEN;
```

Figure 2 ADLEN substructure. C declaration

2.1.4 Offset-Length (OFLEN)

The OFLEN substructure, shown in Figure 3 on page 8, is used only in the EXID structure (see Figure 21 on page 22) to redefine subfields of the identifier string. The field OFLEN.REL counts the number of bytes to skip from the address given by ADLEN.P to reach the described subfield. An OFLEN.LEN value of 0 means the OFLEN.REL field is immaterial.

```
typedef struct {         /* to access to a subfield */
   unsigned int rel;     /* Relative offset of the subfield within adlen.p */
   unsigned int len;     /* Subfield length */
}OFLEN;
```

Figure 3 OFLEN substructure. C declaration

2.1.5 Position (POS)

This substructure is used to report a position in the input stream. POS.ROW and POS.COL are respectively the line and the column in the current entity. POS.OFS is the offset regarding to the beginning of the current entity and POS.ETY_P is a pointer to the ETY structure describing the current entity (See Figure 17 on page 19 for a description of this structure).

```
typedef struct {          /* to describe position in input stream */
   int row;               /* RE count: line number */
   int col;               /* Offset within last row */
   long ofs;              /* Offset from the beginning */
   ETY *ety_p;            /* ETY pointer */
}POS;
```

Figure 4 POS substructure. C declaration

2.2 Parser/Application Communication structure (PAC)

Communication between the Parser and the Application is by a common structure named the PAC. The PAC is built by the Application and given to the Parser on the initial call to the main Parser Entry point. This single PAC structure will be used whenever the Parser and the Application communicate.

It is the responsibility of both the Parser and the Application to remember where the PAC is at all times. Many schemes for doing this rely on specifics of a particular Operating System. To avoid such dependencies, the PAC address is passed to both entry points as the single parameter of a regular C function:

```
function(PAC * pac_p); /* pac_p is a pointer to a PAC structure */
```

For applications that do not follow the C convention, a few lines of code (generally assembler) might be required to do the right convention translation. See Figure 5 for a description of the PAC.

```
typedef struct t_PAC {
   char  id[8];       /* <PAC015>, I'm a PAC */
   short resv;        /* (Reserved, must be zero) */
   short argc;        /* (Reserved, must be zero) */
   char **argv;       /* (Reserved, must be zero) */
   int   (C_FCT *ase)(struct t_PAC *);  /* E.P. Addr for Appl. services */
   int   (C_FCT *pse)(struct t_PAC *);  /* E.P. Addr for Parser services */
   PFC   func;        /* PFC code (See equates) */
   PSF   psf;         /* PSF code (See equates) */
   OEB  *oeb_p;       /* Opened Entity Block (of current entity) */
   AHL   ahl;         /* Application Handle (Identifier or Access) */
   ADLEN dad;         /* Adlen for passed data */
   ADLEN stg;         /* Adlen for storage allocation */

   /* Other Fields */
   POS   pos;         /* Position of the most recently reported event */
   SDC   sdc;         /* SDC code for SKIP_PFC (see equates) */
   int   errcod;      /* Error Code for these series of messages */
   int   msgcod;      /* Message code of this message in the series */
   short errsev;      /* Error Severity of the series of messages */
   short msglast;     /* last message in the series or not (0 or 1) */

   /* Fields for private System (interface Parser <-> Appl) use */
   int   syssa;       /* System private use save word */
   int   syssy;       /* System private use sync. word */
   int   syssv[16];   /* System private save area */
   void *stack_p;     /* Usually: address of parser stack storage */
   void *swork_p;     /* Usually: copy of ase(), when trapping ase's */

   /* Fields for private Parser use */
   void *gwork_p;     /* Work area for use of parser */

   /* Fields for private Application use */
   void *user_p;      /* Usually: regain a reentrancy anchor in services */
}PAC;
```

Figure 5 PAC structure. C declaration of the Parser/Application Communication block

2.2.1 Application function services and notifications codes

Whenever the Parser calls the Application service entry point (the address of which is in the PAC.ASE field of the PAC), it must also request a function by placing the appropriate function code number in the PAC.FUNC field of the PAC. Figure 6 on page 10 lists the values of these function codes.

```
typedef enum {
   RES_PFC=0,        /*  0 is Reserved */
   ELTST_PFC,        /*  1 Start element: pac.dad.p -> DED */
   ELTND_PFC,        /*  2 End element: pac.dad.p -> DED */
   RE_PFC,           /*  3 Relevant record end (No data) */
   TEXT_PFC,         /*  4 Text data: pac.dad */
   PI_PFC,           /*  5 Processing instruction: pac.dad */
   NSGML_PFC,        /*  6 Report Non SGML characters */
   ETY_PFC,          /*  7 Data Entity: pac.dad.p -> ETY */
   MSG_PFC,          /*  8 Message: pac.dad, pac.errc */
   SGET_PFC,         /*  9 Get storage: pac.stg.len; returns pac.stg.p */
   SFRE_PFC,         /* 10 Free storage: pac.stg */
   /* This function takes an External Identifier, returns a Fileid-AHL */
   FIND_PFC,         /* 11 Get a FileId-AHL: pac.dad.p -> EXID */
   /* The following functions take a FileId-AHL, returns an Access-AHL */
   XENOP_PFC,        /* 12 Open External Entity */
   UTOPR_PFC,        /* 13 Open Utility File for Reading */
   UTOPW_PFC,        /* 14 Open Utility File for Writing */
   /* The following functions use an Access-AHL */
   UTWRT_PFC,        /* 15 Write Utility File record: pac.dad */
   UTRD_PFC,         /* 16 Read utility file record: returns pac.dad */
   UTCL_PFC,         /* 17 Close an Utility File */
   XENRD_PFC,      /* 18 Read Data from an External Entity: returns pac.dad */
   XENCL_PFC,        /* 19 Close an External Entity */
   /* Miscellaneous Notifications */
   PROST_PFC,        /* 20 Begin Prolog */
   PROND_PFC,        /* 21 End of Prolog found */
   DOCND_PFC,        /* 22 End of DOC found - Parse Ended */
   ETYST_PFC,        /* 23 Report start parsing an Entity: pac.dad.p -> ETY */
   ETYND_PFC,        /* 24 Report end of an Entity */
   SR_PFC,           /* 25 A mapped shortref has been found */
   MAP_PFC,          /* 26 USEMAP declaration in the instance */
   SKIP_PFC,         /* 27 Skipped data: pac.sdc has the SDC */
   DECST_PFC,        /* 28 Start of SGML Declaration */
   DTDST_PFC,        /* 29 Start of DTD */
   DTDND_PFC,        /* 30 End of DTD */
   DAPND_PFC,        /* 31 End of DOC right After Prolog - Parse continues */
   UNUSED32_PFC,     /* 32 -- no more used -- */
   LTDST_PFC,        /* 33 Start of LTD */
   LTDND_PFC,        /* 34 End of LTD */
   /* GETPOS / SETPOS (only used for LINK) */
   XENGP_PFC,     /* 35 Get position: returns pac.dad=block,length of block */
   XENSP_PFC,     /* 36 Set position: passed pac.dad=block,length of block */
   /* Markup declarations */
   ETYDCLST_PFC,     /* 37 ENTITY declaration start */
   ETYDCLND_PFC,     /* 38 ENTITY declaration end */
   NOTDCLST_PFC,     /* 39 NOTATION declaration start */
   NOTDCLND_PFC,     /* 40 NOTATION declaration end */
   ELTDCLST_PFC,     /* 41 ELEMENT declaration start */
   ELTDCLND_PFC,     /* 42 ELEMENT declaration end */
   ATLELTDCLST_PFC, /* 43 ATTLIST declaration start for element(s) */
   ATLELTDCLND_PFC, /* 44 ATTLIST declaration end for element(s) */
   ATLNOTDCLST_PFC, /* 45 ATTLIST declaration start for notation(s) */
   ATLNOTDCLND_PFC, /* 46 ATTLIST declaration end for notation(s) */

   USER_PFC = 200 /* 200 and higher are reserved for Application own's use */
}PFC;
```

Figure 6 PFC values. application function services and notifications codes

2.2.2 Parser service function codes

Whenever the Application calls the Parser service entry point (the address of which is in the PAC.PSE field of the PAC), it must also request a function by placing the appropriate function code number in the PAC.FUNC field of the PAC. Figure 7 on page 11 lists the values of these function codes.

```
typedef enum {
   RES_PSF=0,     /*  0 is Reserved */
   UNUSED01_PSF, /*  1 -- no more used -- */
   QELT_PSF,      /*  2 Query for an element: pac.dad.p -> ELT */
   QLELT_PSF,     /*  3 List all elmts: pac.dad.p->ELT array, len = count */
   QGETY_PSF,     /*  4 Query for a gen entity: pac.dad.p -> ETY */
   QLGETY_PSF,    /*  5 List all geties: pac.dad.p->ETY array, len=count */
   QPETY_PSF,     /*  6 Query for a prm entity: pac.dad.p -> ETY */
   QLPETY_PSF,    /*  7 List all peties: pac.dad.p->ETY array, len=count */
   QDLM_PSF,      /*  8 Returns the Delimiter Table (dstab) */
   QCAPA_PSF,     /*  9 Returns the capacities tables */
   QETY_PSF,      /* 10 Query the current entity */
   UNUSED11_PSF, /* 11 -- no more used -- */
   UNUSED12_PSF, /* 12 -- no more used -- */
   UNUSED13_PSF, /* 13 -- no more used -- */
   QSR_PSF,       /* 14 Query ShortRef delimiter table */
   QQTY_PSF,      /* 15 Query the declared quantities */
   QOPT_PSF,      /* 16 Query the enabled features/options */
   QFCT_PSF,      /* 17 Query the functions characters */
   QNOTA_PSF,     /* 18 Query a given notation: pac.dad.p -> NCB */
   QLNOTA_PSF,    /* 19 List all notations: pac.dad.p->NCB array, len=count */
   FGETY_PSF,     /* 20 Get access to a gen entity: pac.dad.p -> ETY */
   ALINK_PSF,     /* 21 Activate a linktype */
   ACONC_PSF      /* 22 Activate a concurrent DTD */
}PSF;
```

Figure 7 PSF values. Parser services function codes

2.3 Element Descriptor structure (DED)

Whenever the Parser reports the start or the end of an element, it passes in PAC.DAD the address of a DED[4], which represents the element. The DED structure (described in Figure 8) points to the element type (ELT) in the DTD, and also has flags to describe some characteristics of this element. It also provides an anchor to a possible chain of ATD (Attribute Descriptor) structures that describe the attributes of this element.

Storage for the DED, the ATD chain, and the values those structures point to, is read-only, and is not reusable after the Application returns from the Parser Notification call. It is up to the application to put these values in a safe place, if it requires it.

2.3.1 DED C structure:

```
typedef struct {
   ELT *elt_p;               /* Element pointer */
   ATD *atd_p;               /* Address of the first ATD or 0 */
   ELT *by_elt_p;            /* If this tag was implied by another tag */
   SRM *srm_p;               /* Current shortref map */
   unsigned short is;        /* Bits Flag */
   unsigned char tagmin;     /* Minimization */
   unsigned char impldby;    /* What implied the tag */
}DED;
```

Figure 8 DED structure. C declaration of an Element Descriptor

[4] This structure was once called the "Document Element Descriptor," hence DED. However, the term Document Element is an ISO keyword, designating the doctype name (see 4.99, p10), so this structure was renamed the "Element Descriptor."

The last entry in the DED block is a pointer to the ELT structure. The ELT structure results from the compilation of the ELEMENT markup declaration of the DTD; it defines all characteristics of the element and contains pointer to other control blocks: MDL (tree of the content model), STS and RKL (stem and rank structures), SRM (short reference mapping), EX (exceptions), DTP (data-tag).

2.3.2 Element Flags (LF)

The DED.IS is a Bits Flag (Boolean), i.e. DED.IS holds a value that results from the sum of each individual components, as indicated in Figure 9 on page 13.

```
#define RE_LF  (unsigned char)0x00  /* A Record End */
#define STG_LF (unsigned char)0x01  /* Start Tag */
#define ETG_LF (unsigned char)0x02  /* End Tag */
#define IMP_LF (unsigned char)0x04  /* Implied (vs. User Initiated) */
#define PEX_LF (unsigned char)0x08  /* Plus Exception (vs. Proper Element) */
#define PMT_LF (unsigned char)0x10  /* Premature End Tag */
#define EMP_LF (unsigned char)0x20  /* End tag with invalid empty content */
#define ECR_LF (unsigned char)0x40  /* Explicit Content Ref: empty content */
```

Figure 9 LF values. the value of the DED.IS flag field results from the sum of selected components in the list above

2.4 Attribute Descriptor structure (ATD)

The Attribute Descriptor structure (ATD) describes data attributes (notation parameters) as well as other attributes (element and link attributes). It is described in Figure 10 on page 15.

The address of an ATD is found in:

- the DED.ATD_P field to describe the first attribute of an element,
- the NCB.ATD.P field to describe the first attribute of a N/S/CDATA external data entity, (the NCB is pointed to by the NAD.NCB.P field of a NAD)

  the ATD.NEXT_P field to describe the second through the nth attribute in a chain of attributes (data or others). **For YASP, the ATDs are grouped in a sequential array, so the next pointer is redundant. The LAST_AF flag indicates the last ATD.** *Removing the NEXT pointer would lead to a NAD different from an ATD.*

The ATD list follows the same order specified by the Document Type Declaration: for a given element of a given document, the number of ATDs will then stay the same.

2.4.1 ATD C structure

```
typedef struct t_ATD {       /* Attribute Descriptor Structure */
   union {
      OFS ofs;               /* offset of the attribute name */
      unsigned char *p;      /* ...then relocated (when loaded) */
   }name;
   unsigned short is;        /* Bits Flag */
   unsigned char kind;       /* see AFK list */
   unsigned char defv;       /* Default Value: see AFV list */
   union {
      ADLEN lit;             /* For regular attributes: literal ADLEN */
      ADLEN desc;            /* or literal descriptor if DESC_AF */
      NCB *ncb_p;            /* For  a Notation attribute: NCB pointer */
      struct {               /* For ENTITY or ENTITIES attributes: */
         ETY **pp;           /* 1)- pointer to an array of pointers to ETY */
         unsigned int count; /* 2)- count of ETY(s) in the array */
      }ety;
   }val;
}ATD;
```

Figure 10 ATD structure. C declaration

2.4.2 Attribute Flags (AF, AFK, AFV)

2.4.2.1 ATD.IS

The ATD.IS is a Bits Flag (Boolean), i.e., ATD.IS holds a value that results from the sum of all individual components, as indicated in Figure 11.

```
#define LAST_AF            1  /* last in chain */
#define LISTED_AF          2  /* belongs to a name (token?) group */
#define BAD_AF             4  /* Bad attribute specified (run time) */
#define EXM_AF             8  /* attribute has been examined (run time) */
#define DESC_AF           16  /* Lit is described by ADLEN 0 (run time) */
#define OMITNAME_AF      512  /* Attribute name was omitted */
#define LINK_AF         1024  /* Is a link attribute */
```

Figure 11 AF values. the value of the ATD.IS flag field results from the sum of selected components in the list above

2.4.2.2 ATD.KIND

The ATD.KIND field describes the declared value of an attribute (see ISO8879, 11.3.3), as one of the enumerated values listed in Figure 12 on page 16. Note that an odd number means "plurality."

```
#define INVALID_AFK        0  /* None = Invalid (for unsolved NADs) */
#define CDATA_AFK          2  /* character data */
#define ENTITY_AFK         4  /* general entity name */
#define ENTITIES_AFK       5  /* general entity name list */
#define ID_AFK             6  /* an ID */
#define IDREF_AFK          8  /* an ID reference */
#define IDREFS_AFK         9  /* an ID reference list */
#define NAME_AFK          10  /* a name */
#define NAMES_AFK         11  /* a name list */
#define NMTOKEN_AFK       12  /* a name token */
#define NMTOKENS_AFK      13  /* a name token list */
#define NOTATION_AFK      14  /* notation or solved NAD */
#define NUMBER_AFK        16  /* a number */
#define NUMBERS_AFK       17  /* a number list */
#define NUTOKEN_AFK       18  /* a number token */
#define NUTOKENS_AFK      19  /* a number token list */
#define GROUP_AFK         20  /* one of a fixed group */
```

Figure 12 AFK values. the ATD.KIND flag field has one of the values listed above

2.4.2.3 ATD.DEFV

The ATD.DEFV field describes the default value of an attribute (see ISO8879, 11.3.4), as one of the enumerated values listed in Figure 13.

```
#define INVALID_AFV        0  /* None = Invalid */
#define IMPLIED_AFV        1  /* Impliable attribute */
#define REQUIRED_AFV       2  /* This attribute is REQUIRED  <= 1 => ID */
#define CURRENT_AFV        3  /* Default attribute is CURRENT value */
#define CONREF_AFV         4  /* Content Reference Attribute */
#define DEFLT_AFV          5  /* >= A default value has been specified */
#define FIXED_AFV          6  /* This default value is fixed */
```

Figure 13 AFV values. the ATD.DEFV flag field has one of the values listed above

2.4.3 Isolating C/S DATA entities in CDATA attribute value specification

2.4.3.1 Principles

- An Application must be able to isolate SDATA and CDATA entities expanded by the Parser in CDATA attribute value. Isolating CDATA is an extension to ESIS recommendations.

- Since only the Parser knows how to remove RS and Ee and to replace RE and SEPCHAR with SPACE, the Parser must still present to the Application the expanded attribute value, with the imbedded entities solved.
- In case of external SDATA and CDATA entities, the Application must be able to find out the associated notation and data-attributes.
- This applies only to attribute value literal of element's attribute the declared value of which is CDATA. For others attributes, whether CDATA or SDATA entities were used is irrelevant (7.9.3 note).

Here is how these goals are achieved:

1. In the ATD structure bit-flags field (adt.is), the DESC_AF bit indicates that at least one portion of the attribute value literal was obtained by the expansion of a CDATA/SDATA entity.
2. In case the DESC_AF bit is on, the ADLEN in the "ATD.VAL" union refers to a DESCRIPTOR of the attribute value literal ("ATD.VAL.LIT") rather than to the attribute value literal itself ("ATD.VAL.DESC").
3. A DESCRIPTOR is an array of ADLEN's. The first one contains the address and the length of the (expanded) attribute value literal. The other ADLEN(s) define the composition of each portion of the literal:
   - For a portion that results from the expansion of an entity, the ADLEN has the address of the entity block (ETY structure) and the length of the expanded portion.
   - For a portion that is plain text, the ADLEN has a NULL address and the length of the text portion.

     NOTES:

     - Accessing the ETY is mandatory, to get (for instance) the associated notation and also if it is CDATA or SDATA
     - The literal length found via the ETY structure may be different (bigger) since, for instance, RS may have been removed.
4. The "ATD.VAL.DESC" field tells the address and the number of ADLEN's the DESCRIPTOR is made of.
5. Data-attributes as well as tag-attributes are concerned by this implementation.

See Figure 14 on page 17 for an example.

2.4.3.2 Example

Figure 14 shows how the example below is represented, allowing the application to isolate the different kind of entities the attribute value literal is made of.

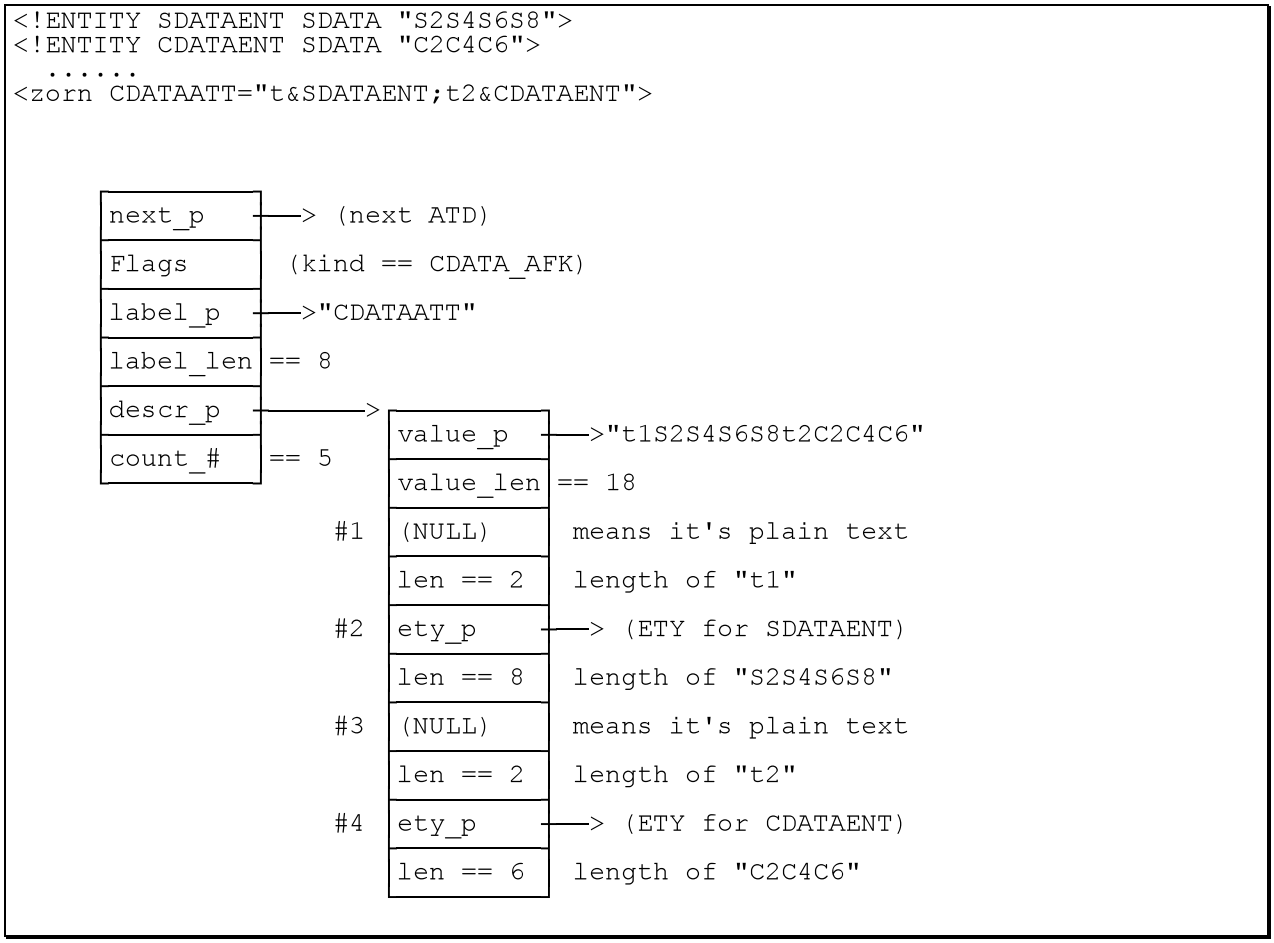

Figure 14 Sample of an ATD structure.

2.5 Notation Descriptor structure (NAD)

The Notation Interpreter Descriptor structure (NAD) describes a Notation Interpreter associated with a non-reparseable CDATA/SDATA/NDATA external entity. It is formally defined by the NAD structure shown in Figure 15.

The address of a NAD will be in the ETY.VAL.EXT.NAD.P field of an external, non reparseable entity, when such an entity is reported. The data attribute specifications on the entity declaration will be accessed through the ATD field.

```
typedef struct {        /* Notation Interpreter Structure */
   union {
      int    ix;        /* index in ATD table */
      struct t_ATD *p;  /* or Attribute Value Specification */
   }atd;
   union {
      int   ix;         /* Index in the NCB array when unloaded or - 1 */
      NCB   *p;         /* After relocation ptr to NCB or 0 if no notation */
   }ncb;
}NAD;                   /* External N/S/CDATA Notation and Attributes */
```

Figure 15 NAD structure. C declaration of a Notation Interpreter Descriptor

2.6 Notation Control Block (NCB)

The Notation Control Block structure (NCB) describes a notation markup declaration and its associated attribute list. The name of the notation is in the EXID.NAME.P field. It is formally defined by the NCB structure shown in Figure 16.

The address of an NCB will be in the NAD.NCB.P field of a Notation Interpreter Descriptor.

```
typedef struct {           /* Notation Control Block */
   EXID exid;
   union {
      int    ix;          /* index in ATD table */
      struct t_ATD *p;    /* or Attribute Value Specification */
   }atd;
}NCB;
```

Figure 16 NCB structure. C declaration of a Notation Control Block

2.7 Entity Descriptor structure (ETY)

Entities are described by the Data Entity Descriptor Structure (ETY), as shown in Figure 17. The address of an ETY[5] is found in:

- the PAC.DAD.P field, when the Parser is reporting an entity directly using the REPORT_A_DATA_ENTITY notification,
- the ATD.VAL.ETY field, when an element's attribute is an entity.

2.7.1 ETY C structure:

```
typedef struct {           /* Entity Descriptor Structure */
   ADLEN name;             /* Item HEADER (SHSH_HDR for shsh hashing) */
   union {
      ADLEN lit;           /* Internal Entity: ADLEN of the literal */
      struct {             /* External entity: */
         union {
            union {        /* 1)- Address of the EXID: */
               int  ix;    /*     - Index in the EXID array */
               EXID *p;    /*     - Pointer (after relocation) */
            }exid;
            int ety_ix;    /* 1)- Index in the ETY array (temporary use) */
         }_;
         union {           /* 2)- External (non reparseable), NAD address: */
            int  ix;       /*     - Index in the NAD array */
            NAD *p;        /*     - Pointer (after relocation) */
         }nad;
      }ext;
   }val;
   unsigned char is;       /* Bits Flag */
   unsigned char kind;     /* value field */
}ETY;
```

Figure 17 ETY structure. C declaration of an Entity Descriptor

2.7.2 Entity Flag (EF, EFK)

2.7.2.1 ETY.IS flag field

The ETY.IS is a Bits Flag (boolean), i.e., ETY.IS holds a value that results from the sum of all individual components, as indicated in Figure 18.

[5] Only NON REPARSEABLE entities are reported: CDATA, SDATA, external NDATA.

```
#define NOTATION_EF       1   /* a notation block is associated */
#define BRACKETED_EF      2   /* must put stuff around it */
#define RCDATA_EF         4   /* RCDATA vs. CDATA/SDATA/NDATA/PI or SUBDOC */
#define EXTERNAL_EF       8   /* external entity vs. memory entity */
#define DEFLT_EF         16   /* it's the default entity */
#define PARAMETER_EF     32   /* it's a parameter entity */
#define USER_EF          64   /* User flag - Init value is "off" */
```

Figure 18 EF values. the value of the ETY.IS flag field results from the sum of selected components in the list above

2.7.2.2 ETY.KIND flag field

The ETY.KIND field describes the kind of Entity (see ISO8879, 10.5), as one of the enumerated values listed in Figure 19.

```
#define INVALID_EFK       0   /* Invalid */
#define CDATA_EFK         1   /* not_rcdata|Ext/Int|notation iff Ext */
#define SDATA_EFK         2   /* not_rcdata|Ext/Int|notation iff Ext */
#define MD_EFK            3   /*   rcdata  |  Int  |   bracketed     */
#define PI_EFK            4   /* not_rcdata|  Int  |   bracketed     */
#define STARTTAG_EFK      5   /*   rcdata  |  Int  |   bracketed     */
#define ENDTAG_EFK        6   /*   rcdata  |  Int  |   bracketed     */
#define MS_EFK            7   /*   rcdata  |  Int  |   bracketed     */
#define NDATA_EFK         8   /* not_rcdata|  Ext  |   notation      */
#define TEXT_EFK          9   /*   rcdata  |  Int  |                 */
#define SGML_EFK         10   /*   rcdata  |  Ext  |                 */
#define SUBDOC_EFK       11   /* not_rcdata|  Ext  |                 */
```

Figure 19 EFK values. the ETY.KIND flag field has one of the values listed above

2.8 External Identifier structure (EXID)

The External Identifier structure is used by the Parser to tell the Application about a specific file. As the Parser is system-independent, it does not know about system-specific file names, and requires the aid of the application to map an EXID to a physical resource within the system.

The EXID structure is filled when the Parser decodes an external identifier (either system identifier or public identifier or both). Figure 20 describes when and how such external identifiers are reported.

Declared as...	EXID Class assigned by YASP[6]	EXID name[7]	System ID, Public ID
(primary entity)	PRIMARY_XFC	N/A	-- None --
<!ENTITY (no type)	TEXT_XFC DOCUMENT_XFC ELEMENTS_XFC ENTITIES_XFC SHORTREF_XFC	entity name	as declared
<!ENTITY ... NDATA	NONSGML_XFC	entity name	as declared
<!ENTITY ... CDATA	NONSGML_XFC	entity name	as declared
<!ENTITY ... SDATA	NONSGML_XFC	entity name	as declared
<!ENTITY ... SUBDOC	SUBDOC_XFC	entity name	as declared
<!DOCTYPE	DTD_XFC	doctype name	as declared
<!NOTATION	NOTATION_XFC	notation name	as declared

[6] See Figure 23

[7] The field EXID.NAME.P is given to help the Application in its strategy to map an EXID to a physical resource.

<!LINKTYPE	LPD_XFC	linktype name	as declared
<!SGML ... CHARSET	CHARSET_XFC	N/A	as declared
<!SGML ... CAPACITY	CAPACITY_XFC	N/A	as declared[8]
<!SGML ... SYNTAX	SYNTAX_XFC	N/A	as declared[8]
Default OCS file	UTSYNL_XFC	N/A	-- None --
User OCS file	UTSYNU_XFC	N/A	-- None --
ODT file (input)	UTODTR_XFC	doctype name	-- None --
ODT file (output)	UTODTW_XFC	doctype name	-- None --

Figure 20 Reporting an External Identifier. Various places where SGML uses an external identifier are reported to the Application according to the values listed.

2.8.1 C structure

```
typedef struct {              /* external resource identifier descriptor */
   ADLEN name;                /* Entity name */
   /*-------------------------------------------+
   |The System Identifier is optional for PUBLIC  |
   |The Public identifier doesn't exist for SYSTEM |
   +-------------------------------------------*/
   ADLEN val;                 /* Identifier */
                              /* Identifier Subfields related to "val" */
   OFLEN sid;                 /* System Identifier */
   OFLEN pid;                 /* Public Identifier */
   /* iff is.VALIDFPI,redefine FPI fields of "pid"(still related to "val") */
   OFLEN ownid;               /* Owner identifier */
   OFLEN txtdesc;             /* Public text description */
   OFLEN lang;                /* lang, or designating seq */
   OFLEN ptdv;                /* Public Text Display Version */
                              /* Flags */
   unsigned short is;         /* Bits Flag */
   unsigned char asgn_cls;    /* Class Field assigned by the Parser */
   unsigned char spec_cls;    /* Class Field specified */
   AHL idhl;                  /* FileId-AHL (when solved) */
}EXID;
```

Figure 21 EXID structure. C declaration of an External Identifier

2.8.2 External Identifier Flag (XF, XFC)

2.8.2.1 XID.IS flag field

The XID.IS is a Bits Flag (boolean), i.e., XID.IS holds a value that results from the sum of all individual components, as indicated in Figure 22. Note that this is a short-type field to handle numbers greater than 256.

[8] System Identifiers are allowed: this is an ISO8879 extension.

```
#define SOLVED_XF      0x0001 /* FIND_PFC has been issued for this XID */
#define INVALID_XF     0x0002 /* FIND_PFC was unable to get a FileId-AHL */
#define VALIDFPI_XF    0x0004 /* PID has a formal structure */
#define AVAILPTEXT_XF  0x0008 /* available public text */
#define PROVPTDV_XF    0x0010 /* System must provide Pub.Text Disp.Vers.*/
#define PTDS_XF        0x0020 /* Pub.Text Design. Sequence vs. Language */
#define ALTERN_XF      0x0040 /* Take alternate system spec if any */
#define DEFLT_XF       0x0080 /* Default entity */
#define REGISTERED_XF  0x0100 /* Owner is registered */
#define ISOREG_XF      0x0200 /* Registration is ISO */
#define PUBLIC_XF      0x0400 /* A Public Identifier was furnished */
#define SYSTEM_XF      0x0800 /* A System Identifier was furnished */
```

Figure 22 XF values. the value of the XID.IS flag field results from the sum of selected components in the list above

2.8.2.2 XID.CLS flag field

The XID.CLS field describes the Public Text Class (see ISO8879, 10.2.2.1) as one of the enumerated values listed in Figure 23. Values greater than 13 are not part of the ISO standard. These values can be redefined by Parsers (in agreement with the Application) to classify a Parser's utility files.

```
#define INVALID_XFC       0   /* Invalid */
#define CAPACITY_XFC      1   /* capacity set */
#define CHARSET_XFC       2   /* character data */
#define DOCUMENT_XFC      3   /* sgml document */
#define DTD_XFC           4   /* document type declaration subset */
#define ELEMENTS_XFC      5   /* element set */
#define ENTITIES_XFC      6   /* entity set */
#define LPD_XFC           7   /* link type declaration subset */
#define NONSGML_XFC       8   /* non sgml data entity */
#define NOTATION_XFC      9   /* character data */
#define SHORTREF_XFC     10   /* short reference set */
#define SUBDOC_XFC       11   /* sgml subdocument entity */
#define SYNTAX_XFC       12   /* concrete syntax */
#define TEXT_XFC         13   /* sgml text entity */
                              /* Parser Specific */
#define UTODTR_XFC       14   /* utility dtd to read (odt) */
#define UTODTW_XFC       15   /* utility dtd to write (odt) */
#define UTSYNU_XFC       16   /* utility user-modified ocs */
#define UTSYNL_XFC       17   /* utility local-system ocs */
#define UTSYNR_XFC       18   /* utility refcon-system ocs */
#define PRIMARY_XFC      19   /* primary entity */
#define UTOLTR_XFC       21   /* utility ltd to read (olt) */
#define UTOLTW_XFC       22   /* utility ltd to write (olt) */
```

Figure 23 XFC values. the XID.CLS field has one of the values listed above

2.9 Element structure (ELT)

This structure (described in Figure 24), is the result from the compilation of the DTD for a specific element.

The address of an ELT structure is found in:

- the last field of the DED structure (ELTST_PFC/ELTND_PFC PFC's),
- PAC.DAD.P on return from the QELT_PSF and QLELT_PSF Parser Services

2.9.1 ELT C structure

```
typedef struct t_ELT {   /* Element Type Definition */
   union {               /* Item HEADER (SHSH_HDR for shsh hashing) */
      OFS ofs;           /* offset of the elt name */
      unsigned char *p;  /* ...then relocated (when loaded) */
   }name;
   ELT_FLAGS is;
   union {               /* -1 if: ANY, EMPTY, DCDATA (RCDATA || CDATA) */
      int  ix;           /* or index in MDL array of the model */
      MDL *p;            /* when loaded, pointer to the model header */
   }mdl;
   union {
      int  ix;           /* index in an ATD array of first attribute value */
      ATD *p;            /* when loaded, points an AD array */
   }atd;                 /* Attribute Definition */
   unsigned int atdsize; /* size of the ad array */
   union {
      int  ix;           /* index in an EX array */
      EX  *p;            /* when loaded, points an EX array */
   }mex;                 /* minus exceptions */
   union {
      int  ix;           /* index in an EX array */
      EX  *p;            /* when loaded, points an EX array */
   }pex;                 /* plus exceptions */
   union {
      int  ix;           /* point an array of indices to ETY array */
      SRM *p;            /* when loaded, points an array of ETY ptrs */
   }srm;                 /* short ref map */
   union {
      int  ix;           /* index in an RKL array (-1 if not ranked) */
      RKL *p;            /* when loaded, points an RKL */
   }rkl;
}ELT;
```

Figure 24 ELT structure. C declaration of the Element structure

The information an Application can find useful are mainly the ELT_FLAGS, the ATD and the SRM sub-structures. The ELT_FLAGS is described in Figure 25, the ATD in Figure 10, and the SRM in Figure 26.

The MDL, EX, STS and RKL sub-structures should be considered to be internal to YASP. However, a description of these structures can be found in the YASPAPI.H file.

2.9.2 ELT_FLAGS C structure:

```
typedef struct {
   unsigned int reqd_open_tag :1; /* Minimization */
   unsigned int reqd_end_tag  :1; /* Minimization */
   unsigned int any          :1; /* No mdl_p: model content: ANY */
   unsigned int empty        :1; /* No mdl_p: dcl content EMPTY (see Note)*/
   unsigned int dcdata       :2; /* No mdl_p: dcl content DCDATA (see above)*/
   unsigned int pcdata       :1; /* Mixed content and data-char allowed */
   unsigned int required     :1; /* An attribute in the list is REQUIRED */
   unsigned int current      :1; /* An attribute in the list is CURRENT */
   unsigned int conref       :1; /* An attribute in the list is CONREF */
   unsigned int notation     :1; /* An attribute in the list is NOTATION */
   unsigned int entity       :1; /* An attribute in the list is ENTITY(IES) */
   unsigned int solved       :1; /* Internal use */
   unsigned int unused       :3; /* for future use... */
}ELT_FLAGS;
```

Figure 25 ELT_FLAGS structure.

2.10 Short Reference Map descriptor (SRM)

This structure (described in Figure 26), is the result from the compilation of the ShortRef Maps.

The address of an SRM structure is found in:

- the ELT.SRM.P field of the ELT structure,
- the PAC.DAD.P for a MAP_PFC notification,
- the PAC.DAD.P for a SR_PFC notification.

```
typedef union {            /* element of a (sr_max+1) array of: */
   union {                 /* Item HEADER (SHSH_HDR for shsh hashing) */
      OFS ofs;             /* offset of the map name */
      unsigned char *p;    /* ...then relocated (when loaded) */
   }mapname;
   unsigned char *name_p;  /* while building it: entity name pointer */
   int  ix;                /* unloaded: indices to an array of ETY */
   ETY  *ety_p;            /* final: ETY pointers */
}SRM;
```

Figure 26 SRM structure. Short Reference Map descriptor

The whole map itself is an array of these SRM structures. The number of elements in the array is the number of Short References as defined in the SGML syntax. The Application can obtain this number thru the QCAPA_PSF Parser Service: See 7.1.8.

2.11 Skipped Data Codes (SDC)

These codes are used by the SKIP_PFC notification. They identify the reason why some data from the original document are skipped and not reported by the parser thru another notification.

```
typedef enum {
   NULL_SDC=0,       /*  0 is reserved */
   COMMENT_SDC,      /*  1 SGML Comment */
   ERROR_SDC,        /*  2 SGML Error: text ignored */
   MSINCL_SDC,       /*  3 Marked Section INCLUDE or TEMP */
   MSIGN_SDC,        /*  4 Marked Section IGNORE <![IGNORE [data]]> */
   MSCDATA_SDC,      /*  5 Marked Section CDATA  <![CDATA [ */
   MSRCDATA_SDC,     /*  6 Marked Section RCDATA <![RCDATA [ */
   MSC_SDC,          /*  7 Marked Section Close: ]]> */
   MDOMDC_SDC        /*  8 Found <!> */
}SDC;
```

Figure 27 SDC values. The value appearing in the PAC.SDC field is one of the values listed above

3. Calling Protocols

This section describes the main parser entry point and specific protocols for each of the Application services provided to the Parser by the Application.

3.1 Main Parser Entry

A Startup code can be required to establish the required environment in exchanging information between the Parser and its Application. The design of this interface minimizes the amount of code required for this "boot_strap." Because the only argument passed to the Parser is the PAC address, and there is a unique entry point to Application Services, the startup code is kept easy to design and maintain[9]. C conventions are used, because it is one of the most popular language for SGML Parsers.

If required, the startup code must:

- allocate a stack
- make the PAC address appear as a C argument to the Parser Main Entry Point
- trap the call to Applications Services to pass the PAC address conforming to what the Application expects and return to the Parser putting the return code where the Parser expects it.

When the Parser is first called, the Application has already filled in the fields that it knows about in the PAC. The Parser may read (but not alter) the following fields:

- PAC.ID contains the string "<PAC015>" to verify that this is a "version 15" PAC structure.
- PAC.ASE contains a pointer to the entry point of the main Application service manager routine. For example, reentrancy must be inherent to the Parser.

The fields:

- PAC.SYSSA, PAC.SYSSY, and PAC.SYSSV are reserved for the Application's own use.
- PAC.STACK_P and PAC.PWORK_P can be used by the startup code to store the address of the stack, and other information. These fields are left unchanged by the Parser itself.
- PAC.GWORK_P is used by the Parser to save a pointer to whatever it wants.

In case the primary entity has no "DOCTYPE" declaration, the Parser will call the Application to find a doctype using the FIND function code, with (see Figure 20):

- the EXID.CLS field being UDODTR_XFC
- the EXID.NAME.P field describing the name of the first element found in the document instance, if and only if the document instance started with an element. Otherwise this EXID.NAME.LEN is zero.

It is up to the Application to map this EXID to the physical resource. For instance, if the EXID.NAME.P field does not correspond to a known document element, the Application can default to another document element. This approach allows a document "fragment"[10] to be properly handled.

NOTE: The primary input file or "primary entity" is not open when the Parser is first called. The Parser must issue the FIND and XENOP functions for this entity as for any other entity. The Parser will close the "primary entity" before its final return.

[9] C Parsers that conform to this Interface must be aware they need to work with a minimal C environment, as all I/O and memory functions are provided by the Applications Services. Those parsers must be designed to avoid a requirement for any special features from the application startup

[10] A document that does not begin with a document element, and that has no DOCTYPE declaration to specify it.

3.2 Application Service Routine Convention

The Parser obtains services from the Application by placing a function code in the PAC.FUNC field and calling the Application service entry, pointed to by the PAC.ASE field.

In addition to the PAC.FUNC field, many Application services require the Parser and the Application to communicate data. The address and length of data passed between the Parser and the Application are usually indicated in the PAC.DAD AdLen field, although some functions require other PAC fields to be filled in. Whoever obtains free storage to hold such data is responsible for freeing it. All calls to the Application Services pass the PAC address as the parameter.

The call is typically a C function call, that can be trapped by the startup code to suit specific linkage conventions. The Application may be aware that the PAC.ASE field could have been changed by the startup code, to set up this trap. The Parser expects a return code passed from the Application as the result of the C function call. Again, the startup code can translate the Application specific convention to the C convention, before returning to the Parser.

If the application returns any data or other information to the Parser, it does so by filling in the appropriate fields in the PAC.

3.3 Global Return Codes

Each function may have its own return codes to indicate error conditions or unusual situations. Some of the specific function return codes merely provide interesting information, and may safely be ignored by the Parser, if the Parser does not care about the condition.

The following return codes are always possible from any function. They are not repeated in each function description below:

- RC 0: Normal, function completed without any errors or exceptional conditions
- RC 10: Any request may return code 10 at any time. RC 10 always means “Abandon Ship.” When any function returns with RC 10, the Parser and the Application must cease operation and not do anything other than required clean-up, such as releasing allocated storage and closing files. Possible causes for RC 10 include:
    - An application procedure executed a “QUIT” command
    - A severe system error occurred
    - The Parser and the Application appear to be out of synchronization. For example, if the Parser passes an invalid PAC pointer, or asks to end an element that the Application believes is not started, or is not the innermost element, then Return Code 10 will result.

4. Parser Notifications of SGML events to the Application

All required SGML applications functions are the notifications to the Application — telling SGML events to the Application. Every Parser that works with this interface must be able to tell the Application about such events.

4.1 Report the Start of an Element

This notification is used to tell the Application that the start of an element has been found. When the Parser calls the Application to start an element, it passes a pointer to a structure called the DED (Element Descriptor, see Figure 8). This DED provides all required information about this element. It also points to a chain of ATD structures (Attribute Descriptor, see Figure 10) that provide information about all attributes of this element.

The ATD blocks appear in the same number and order as that of the attribute definitions in the DTD. As usual, the Parser must have completed all minimization.

In no case can the Application change any structure passed by the Parser. Also, it is up to the Application to save any structure it intends to use after it returns to the Parser. The Parser can reuse these fields for its own purpose.

The following fields in the PAC must be set by the Parser prior to calling the service routine to report the start of an element:

- PAC.DAD.P Pointer to the DED describing the element
- PAC.FUNC ELTST_PFC function code

Return codes:

RC 4: Application does not recognize this element (unmapped)

NOTE: The Application has issued its own message. The unknown element is still started, so it may be ended; But too many of these exceptions could indicate a wrong procedure library or some such problem.

4.2 Report the End of an Element

This notification is used to tell the Application the end of an element has been found. (For empty elements, this call will immediately follow the “Start Element” call.)

When the Parser asks the Application to end an element, the element to be ended must be the innermost element.

The Application might be able to figure out for itself what the element is, but nevertheless, this interface requires the Parser to pass a pointer to the innermost element’s DED so the Application can check that it agrees with the Parser, and it is in synch with the Parser.

However, in the DED of this element, only the DED.ELT.NAME ADLEN field is relevant.

The following fields in the PAC must be set by the Parser prior to calling the service routine to report the end of an element:

- PAC.DAD.P Pointer to the DED describing the element
- PAC.FUNC ELTND_PFC function code

4.3 Report a Relevant Record End

The Parser reports relevant Record-ends to the Application via this function, as a separate call. The Parser must report a relevant record end after having reported to the Application whatever came before that Record end. This will generally be Text or End an element.

The rules for determining relevant record ends are formally defined in Section 7.6.1 of ISO 8879.

The Parser sets the PAC.FUNC field to RE_PFC prior to calling the service routine to report a relevant record end:

4.4 Pass Text Data

This notification is used to pass a piece of text to the Application. This call will not be used in case the resulting text is the replacement text of a CDATA/SDATA/NDATA external entity. Instead, the "Report a Data Entity" notification will be used.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD AdLen describing the address and length of the data
- PAC.FUNC TEXT_PFC function code

4.5 Pass a Processing Instruction

This notification is used to execute a Processing instruction. The parser will report all PI through this notification, even if the PI results from the replacement of an entity (instead of using the "Report a Data Entity" notification).

Processing instructions have meaning defined by the Application.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD AdLen describing the address and length of the processing instruction
- PAC.FUNC PI_PFC function code

Return codes:

RC 4: Application does not recognize this processing instruction. It is undefined or unmapped.

NOTE: The Application must have issued its own message in this case.

4.6 Report Non-SGML character

The Parser reports invalid Non-SGML characters to the Application via this function, as a separate call. The Parser must report Non-SGML characters after having reported to the Application whatever came before that Non-SGML character. This will generally be Text or End an element or another Non-SGML character.

- PAC.DAD.P Points to the Non-SGML character that caused the call
- PAC.DAD.LEN Is "1," to later allow more then one character to be reported
- PAC.FUNC NSGML_PFC function code

4.7 Report a Data Entity

The Parser does not need to parse entities that contain data that is known to be impossible or useless to parse. Such entities may contain CDATA, SDATA and NDATA. The Parser passes such entities to the Application using this function.

The Parser tells the Application about a data entity by providing a pointer to an ETY (see Figure 17).

- For an internal entity (CDATA or SDATA), the ADLEN field ETY.VAL.LIT provides the address of the entity value literal.
- For an external entity (CDATA, SDATA or NDATA), the field ETY.VAL.EXT._.EXID.P provides a pointer to an EXID structure for the external entity.

The Parser must use the I/O function to get such a handle, passing the EXID structure that describes the external entity (see GET_A_FILEID function). External entities (except SUBDOC) passed by this function are not reparseable and a Notation is always associated. The field ETY.VAL.EXT.NAD.P points to a NAD (Notation Interpreter Descriptor, see Figure 15).

NOTE: As said before, entities whose type is PI are reported through the PI_PFC function code. However, since the Parser reports the start and end of all entities (including the internal ones), with the ETYST_PFC/ETYND_PFC function codes, an application that needs to known that the PI originates from an entity can access this information.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to the ETY structure processing instruction
- PAC.FUNC ETY_PFC function code

Return codes:

- RC 4: Application does not recognize the given SDATA.

  NOTE: The Application must have issued its own message

- RC 8: Unrecognized data entity type.

Undeclared entities

Ordinarily, it is up to the Parser to determine what to do about an undeclared entity. However, ISO 8879 provides a mechanism for defining a default entity whose value will be substituted in place of any undeclared general entity reference. The default entity can be declared the same as any other general entity. If an undeclared general entity reference is found, and the Parser has a default entity declaration, then the Parser will call this function. The ETY structure will have the ETY.IS.DEFAULT flag on to indicate that the value is the value of the default entity, and the ETY.NAME field will describe the name of the undeclared entity (the default entity has no useful name associated as it uses a reserved name).

4.8 Report the Start of an Entity

The parser reports to the application that a new entity is about to be read. This notification may NOT be in synchronization with the report of the event flow.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to the ETY structure
- PAC.FUNC ETYST_PFC function code

4.9 Report the End of an Entity

The parser reports to the application that a new entity is about to end. This notification may NOT be in synchronization with the report of the event flow.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to the ETY structure
- PAC.FUNC ETYND_PFC function code

4.10 Report a USEMAP in the Document Instance

The parser reports that a USEMAP has been found in the document instance.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to the SRM structure
- PAC.FUNC MAP_PFC function code

Notes:

1. For the #EMPTY map, the map name in the SRM structure is a null string.
2. The application can easily find any short reference map "automatically" activated by an element (i.e USEMAPped in the DTD). Briefly: the DED structure has a pointer to the ELT structure, the ELT structure has a pointer to the SRM structure.

4.11 Report a Mapped Shortref

The Parser reports that a shortref mapped to an entity has been found.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to the SRM structure
- PAC.DAD.LEN Shortref number
- PAC.FUNC SR_PFC function code

Notes:

1. this doesn't means the entity is contextually valid. If it is, one of the ETYST_PFC or ETY_PFC PFC's will follow. If it isn't, an error message will be issued.
2. The SR number indicates the relative position of the SR within the SHORTREF parameter of the SGML declaration. It is also the index to the ETY structure in the SRM. For the concrete syntax, SR#6 is, for instance, "Blank record".

4.12 Report that some Data were Skipped

The parser reports that some significant data has been skipped. "Significant" means data that are neither blanks or Record End/Start. The reason why the data were skipped is given in the PAC.SDC field of the PAC. See Figure 27 for a description of the SDC codes.

The Parser sets the PAC.FUNC field to SKIP_PFC prior to calling the service routine to report that some data were skipped.

NOTE: The parser doesn't give any information about the content of the Skipped Data. If the application needs it, it should rely on the PAC.POS field of the PAC to retrieve the data from the original entity.

4.13 Report the Beginning of Prolog

The Parser reports the beginning of the prolog. This notification is particularly useful for SUBDOC's. Most of the Parser Services cannot be used at this point.

The Parser sets the PAC.FUNC field to PROST_PFC prior to calling the service routine to report that the prolog begins.

4.14 Report the End of Prolog

The Parser reports the end of the prolog. All Parser Services are available at this point.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.FUNC PROND_PFC function code
- PAC.DAD.P Points to the base-element ELT structure

4.15 Report the End of the Document

The Parser reports the end of the document. This notification is issued for the main document instance as well as for SUBDOC's if any.

NOTE: This is the last notification before the parser dies. It can be used to keep the parser in memory and to issue some Parser Services: on return from this notification, the parser will exit.

The Parser sets the PAC.FUNC field to DOCND_PFC prior to calling the service routine to report the end of the document.

4.16 Report the Start of the SGML declaration

The Parser reports the start of the SGML declaration. At this point all Parser Services are unavailable.

The Parser sets the PAC.FUNC field to DECST_PFC prior to calling the service routine to report the start of the SGML declaration.

4.17 Report the Start of the DTD

The Parser reports the start of the DTD. At this point the Parser Services for DTD components are unavailable.

The Parser sets the PAC.FUNC field to DTDST_PFC prior to calling the service routine to report the start of the DTD.

4.18 Report the End of the DTD

The Parser reports the end of the DTD. At this point the Parser Services for DTD components are unavailable.

The Parser sets the PAC.FUNC field to DTDND_PFC prior to calling the service routine to report the end of the DTD.

4.19 Report the Start of an ENTITY Declaration

The Parser reports the start of an ENTITY Declaration. At this point the Parser Services for DTD components are unavailable.

The Parser sets the PAC.FUNC field to ETYDCLST_PFC prior to calling the service routine to report the start of the ENTITY Declaration.

Reporting the starts and ends of markup declarations may help an application to handle parameter entity références used inside or outside these declarations.

4.20 Report the End of an ENTITY Declaration

The Parser reports the end of an ENTITY Declaration. At this point the Parser Services for DTD components are unavailable.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to the ETY structure
- PAC.FUNC ETYDCLND_PFC function code

In the ETY structure the following fields may be accessed :

- name, is, kind
- val.lit, val.ext._.exid

Any attempt to access other fields is not significant.

4.21 Report the Start of a NOTATION Declaration

The Parser reports the start of a NOTATION Declaration. At this point the Parser Services for DTD components are unavailable.

The Parser sets the PAC.FUNC field to NOTDCLST_PFC prior to calling the service routine to report the start of the NOTATION Declaration.

4.22 Report the End of a NOTATION Declaration

The Parser reports the end of a NOTATION Declaration. At this point the Parser Services for DTD components are unavailable.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to the NCB structure
- PAC.FUNC NOTDCLND_PFC function code

In the NCB structure the following fields may be accessed :

- exid

Any attempt to access other fields is not significant.

4.23 Report the Start of an ELEMENT Declaration

The Parser reports the start of an ELEMENT Declaration. At this point the Parser Services for DTD components are unavailable.

The Parser sets the PAC.FUNC field to ELTDCLST_PFC prior to calling the service routine to report the start of the ELEMENT Declaration.

4.24 Report the End of an ELEMENT Declaration

The Parser reports the end of an ELEMENT Declaration. At this point the Parser Services for DTD components are unavailable.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to an array of ELT structures
- PAC.DAD.LEN Number of ELT structures in the array
- PAC.FUNC ELTDCLND_PFC function code

In the ELT structures the following fields may be accessed :

- name, is

Any attempt to access other fields is not significant.

4.25 Report the Start of an ATTLIST Declaration for elements

The Parser reports the start of an ATTLIST Declaration for element(s). At this point the Parser Services for DTD components are unavailable.

The Parser sets the PAC.FUNC field to ATLELTDCLST_PFC prior to calling the service routine to report the start of the ATTLIST Declaration.

4.26 Report the End of an ATTLIST Declaration for elements

The Parser reports the end of an ATTLIST Declaration for element(s). At this point the Parser Services for DTD components are unavailable.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to an array of ELT structures
- PAC.DAD.LEN Number of ELT structures in the array
- PAC.FUNC ATLELTDCLND_PFC function code

In the ELT structures the following fields may be accessed :

- name, atd

Any attempt to access other fields is not significant.

4.27 Report the Start of an ATTLIST Declaration for notations

The Parser reports the start of an ATTLIST Declaration for notation(s). At this point the Parser Services for DTD components are unavailable.

The Parser sets the PAC.FUNC field to ATLNOTDCLST_PFC prior to calling the service routine to report the start of the ATTLIST Declaration.

4.28 Report the End of an ATTLIST Declaration for notations

The Parser reports the end of an ATTLIST Declaration for notation(s). At this point the Parser Services for DTD components are unavailable.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD.P Points to an array of NCB structures
- PAC.DAD.LEN Number of NCB structures in the array
- PAC.FUNC ATLNOTDCLND_PFC function code

In the NCB structures the following fields may be accessed :

- exid.name, atd

Any attempt to access other fields is not significant.

5. Parser Requests for Application Services

5.1 Message Delivery

The Application must provide a Service to deliver message: this permits a more versatile implementation within a better integrated system. For instance, an Application may deliver messages to the same destination as its other messages, or filter messages, or stop the run if the message is more severe than an user-requested threshold. This is made possible if the Application controls messages from the Parser. Also, this implementation takes into account National Language Support requirements.

5.1.1 Principles

An error or other reportable condition can generate more than one message. All the messages associated with one condition are called a "series of messages." A message series has at least one main message typical of the error, and optionally, other general-purpose messages, such as an indication of the location of the error, a message that the Parser terminated, etc. When the parser detects an error, it will pass the "series of messages" to the interface by issuing as many calls as the number of messages in the series.

For all calls related to the same "series of messages," the error code and the error severity will stay the same. Each message is identified uniquely by a value contained in a message identifier field. The last message of a given series is flagged. This allows differentiation of two series of messages with the same error code that follows each other.

A message is composed of fixed fields that are language dependent associated with variable fields that are language invariant (such as tag names, etc..). The value of each variable field (language invariant and known only to the Parser) is passed to the application using an array of ADLEN's, since the length of each variable field can vary. These variable fields must be inserted at different places within the fixed fields, depending on the language used. For a given message, the variable fields is defined by a specific and unique codification: a '%' character followed by a number[11]. The number indicates the position within the array of ADLEN of the message (first ADLEN is 1, and so on).

The number of variable fields related to a given message depends on the message itself: the array of ADLEN's is described by an ADLEN in PAC.DAD.

The Parser is deemed to furnish a list of message identifiers describing for each message:

- the meaning of the message
- an ordered list of all variables fields passed through the tables and their meaning
- the error code, if the message is typical of the error (or an indication that the message is for general purposes)[12].

The documentation of the parser must furnish a detailed description of each error code to allow the application to build a help file. General purpose messages must be "self-explanatory".

5.1.2 Display a Message

This service is used to pass a message to the Application.

Messages are typically delivered in a series, or suite, of messages, with the Parser providing variable information to be substituted into the base message text. The Application may suppress certain messages at its discretion.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD AdLen describes a list of variables to be plugged into message text.

[11] In order to represent a "real" '%' character, the message itself must contains 2 percent character ("%%").

[12] With YASP, the error code of an error-typical message has the same value of its message identifier.

- PAC.ERRCOD The identifier of this suite of messages
- PAC.ERRSEV The severity code for this suite of messages
- PAC.MSGCOD The identifier of this particular message in the suite
- PAC.MSGLAST True if this is the last message in the suite PAC.FUNC Function code &PACMSG..

The Parser should set the severity indicator in PAC.ERRCOD as follows:

- 0 (I) Informational message (Never exceeds AutoQuit)
- 4 (W) Warning
- 8 (E) Error (something is bad, but Parser can recover)
- 12 (S) Severe error (probably can't recover from this)
- 16 (T) Terminating error (Must get RC 10 back from this!)

The message function always returns either 0 or 10, according to whether the severity indicator in PAC.ERRCOD exceeded the Application's "AutoQuit" threshold or not.

5.2 Storage Management Function

The Application must provide a Service to obtain and free storage: due to the various kind of memory management within systems and CPU, the Application will adopt the best strategy. It is recommended that the Parser reduces the number of Storage Management calls and issue these only for a "substantial" amount of memory.

5.2.1 Obtain Storage

This service is used to allocate a piece of memory for use by the Parser. The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.STG.LEN Contains the size of the piece of memory to allocate.
- PAC.FUNC Function code SGET_PFC.

Upon return from the service routine, the return code will be zero if the memory was allocated, otherwise it will be 10. In addition, the following PAC fields will be set:

PAC.STG.P contains a pointer to the new piece of memory.

5.2.2 Return Storage

This service is used to free a piece of memory that was allocated by the Parser. The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.STG.P A pointer to the piece of memory to free.
- PAC.STG.LEN Contains the size of the piece of memory to free.
- PAC.FUNC Function code SFRE_PFC.

  NOTE: The piece of storage to be freed must be the same length as when it was originally allocated. Partial pieces of storage may not be freed.

If the Parser tries to free storage that it did not get, this will result in Return code 10.

5.3 I/O FUNCTIONS

The term "file" designates a set of data[13] that must be accessed using the I/O functions provided by the System. A file will generally reside on external media (disk, tape). However, its physical location is irrelevant for the Parser and the System could use whatever is appropriate: can be a UNIX environment string, or data associated to an EXEC PARM in MVS...

1. SGML files are regular files that can be SGML-defined on any SGML system. These are of two kinds:
   - External Entities, the Parser may need to read
   - Notation Interpreters, the Parser may need to refer to
2. Utility files are files used internally by the Parser code to store data and retrieve it later. The Interface provides a separate set of I/O functions to handle these Utility Files.

To each "File" (External Entity, Notation Interpreter or Utility File) is associated an External Identifier structure (EXID, see Figure 21). For SGML files the EXID is a translation of the external identifier (as defined at 10.1.6 in ISO 8879). For Utility files an EXID is created by the Parser, with enough information to allow the Application to map it to a convenient System File, as shown by Figure 20.

In order to refer to a file, the Parser first requires a FileId-AHL from the Application: this is where the Application can "cook" a true System-File-Identifier from the passed EXID (function GET_A_FILEID). However, a FileId-AHL does not mean the file is opened (it can be impossible to do so, i.e., for Notation Interpreter). To be able to read a File, the Parser requires the Application to Open the File, passing the FileId-AHL. The Application will then return an Access-AHL that will be used by the Parser to really access the contents of the File.

As Notation Interpreter files cannot be opened, three kinds of Open Functions have been defined:

- Open an External Entity (Read Only)
- Open a Utility File for Reading
- Open a Utility File for Writing

Other I/O functions are:

- Read from an External Entity
- Read from a Utility File
- Write to a Utility File
- Close an External Entity
- Close a Utility file

Providing two sets of I/O functions for External entity and utility file allows the Application to process utility files in a different way than SGML files. For utility files less services are required from the Application:

- Utility files are created by the Parser, so the Parser knows in advance how they look. As a result, there is no need for the Application to provide a buffer for these files (see below).
- no RE/RS character handling is required[14].

The Parser writes utility files by "bunch of bytes," a Record-Oriented system can consider as records. The Parser knows in advance the size of a record, or at least, of a set of records. Therefore, the Application does not need to buffer reads to utility files. The Parser is able to furnish a buffer of exactly the required size, as it

[13] "data" must be understood as "a bunch of 0's and 1's": it can contain information, as well as computer code

[14] In C jargon, utility files are "binary files," while external entity are "text files."

knows the size of the record, or the set of records it wants to get. For utility files, the buffer belongs to the Parser. For external entities, the buffer belongs to the Application.

5.3.1 Get a FileId

The Parser uses this function to get a FileId-Handle for any file (External Entity, Notation Interpreter or Utility File). The Parser provides an External Identifier structure (EXID, see Figure 21) that contains all the information it knows about the file, including the entity name, a possible system identifier and a public identifier. The Application will then return a FileId-AHL that resulted from "cooking" the information passed.

A FileId-AHL cannot be directly used to read data. Instead, the "OPEN" I/O function must be issued first, delivering the Access-AHL, i.e., the key to read data.

The following fields in the PAC must be set by the Parser prior to calling the service routine to get a FileId-AHL:

- PAC.DAD.P Points the EXID structure
- PAC.FUNC FIND_PFC function code

Upon normal return, the Application has set the FileId-AHL in the PAC.AHL field.

Return Codes:

- RC 2: Warning: external entity is already opened
- RC 4: Could not associate a FileId-AHL

5.3.2 Open an External Entity

The Parser uses this function to get an Access-AHL to an external entity, and also open it for reading. The Parser provides a FileId-AHL, as described in the previous section. The Access-AHL will be used to read the file.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.AHL The FileId-AHL
- PAC.FUNC XENOP_PFC function code

Upon normal return, the Application has set the Access-AHL in the PAC.AHL field.

Return Codes:

- RC 4: File is already opened
- RC 6: Cannot open (ex: UNIX exclusive access)
- RC 8: Invalid FileId-AHL

5.3.3 Open a Utility File for Writing

The Parser uses this function to get an Access-AHL to a utility file, and also open it for writing. The Parser provides a FileId-AHL. The returned Access-AHL will be used later to write the file.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.AHL The FileId-AHL
- PAC.FUNC UTOPW_PFC function code

Upon normal return, the Application has set the Access-AHL in the PAC.AHL field.

Return Codes:

- RC 2: File already exists (but Access-AHL is OK)

- RC 4: File is already opened
- RC 6: Cannot open
- RC 8: Invalid FileId-AHL

5.3.4 Open a Utility File for Reading

The Parser uses this function to get an Access-AHL to a utility file, and also open it for reading. The Parser provides an FileId-AHL. The Access-AHL will be used to read the file.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.AHL The FileId-AHL
- PAC.FUNC UTOPR_PFC function code

Upon normal return, the Application has set the Access-AHL in the PAC.AHL field.

Return Codes:

- RC 4: File is already opened
- RC 6: Cannot open
- RC 8: Invalid FileId-AHL

5.3.5 Read Data from an External Entity

This service is used to read "some data" from an External Entity. The amount of data read is Application dependent. This scheme is well adapted to any I/O system, either stream I/O, or Record Oriented.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.AHL The Access-AHL
- PAC.FUNC XENRD_PFC function code

Upon normal return, the Application has set the length and a pointer to the Data read in the PAC.DAD field.

Return Codes:

- RC 0: No RS/RE need to be inserted by the Parser
- RC 1: RS/RE character must be inserted at the beginning/end of the record (i.e., the Application read a whole record, but has not generated the RS/RE character). Two extra-bytes at (buffer-1) and (buffer + length) are available for the PARSER, if it needs to force the RS/RE Characters.
- RC 2: An RS must be inserted at the beginning of the Record
- RC 3: An RE must be inserted at the end of the Record
- RC 4: End of File was reached before data could be read
- RC 8: Invalid Access-AHL

5.3.6 Write a Record to a Utility File

This service is used to write a record (variable length) to a Utility File. The Parser is deemed to later read the file the same way it was written (i.e., in "records").

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.AHL The Access-AHL
- PAC.DAD Data to be written (length of and pointer to)

- PAC.FUNC UTWRT_PFC function code

Return Codes:

RC 8: Invalid Access-AHL

5.3.7 Read Record(s) from a Utility File

This service is used to read record(s) from a Utility File into a buffer provided by the Parser, the address of which is found in the PAC.DAD.P field. The number of records to read results from the size of the buffer, as indicated in the PAC.DAD.LEN field. The Application reads as many records from the utility file as required to fill the buffer up. The buffer must exactly fit. Otherwise an error return code is issued, generally indicating the Utility File is not the one expected.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.DAD Data buffer (length of and pointer to)
- PAC.AHL The Access-AHL
- PAC.FUNC UTRD_PFC function code

Return Codes:

- RC 4: Invalid size
- RC 8: Invalid Access-AHL

5.3.8 Close an External Entity

This service is used to close an External Entity (file) that was previously opened by the Application. The Parser must identify the file to be closed by passing its Access-AHL in the PAC.AHL field.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.AHL The Access-AHL
- PAC.FUNC XENCL_PFC function code

Return Codes:

RC 8: Invalid Access-AHL

5.3.9 Close a Utility File

This service is used to close a Utility File that was previously opened by the Application. The Parser must identify the file to be closed by passing its Access-AHL in the PAC.AHL field.

The following fields in the PAC must be set by the Parser prior to calling the service routine:

- PAC.AHL The Access-AHL
- PAC.FUNC UTCL_PFC function code

Return Codes:

RC 8: Invalid Access-AHL

6. Parser Services

The Parser Services provide advanced functions to an Application. When an Application has control, typically to process a Notification service, it may in turn request a Parser service. Parser Services regarding the SGML declaration are not available until the Prolog Start notification. Parser Services regarding the document's DTD are not available until the Prolog End notification.

6.1 Query Element

An Application may use this function to cause the Parser to return an ELT structure. To use this function, the Application must provide the name of the wanted element.

The following fields in the PAC must be set by the Application prior to calling the service routine to query an element:

- PAC.DAD.P Pointer to an ASCIIZ string containing the element name
- PAC.FUNC QELT_PSF function code

Upon normal return, PAC.DAD.P points to the ELT structure for this element

Return Codes:

- RC 2: Invalid context

  This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.

- RC 4: Element not in DTD (unknown)

6.2 Query the Full List of Elements

An Application may use this function to cause the Parser to return a list of all the ELT structures.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the list of elements:

- PAC.FUNC QLELT_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to an array of ELT structures
- PAC.DAD.LEN Number of ELT structures in the array

Return Codes:

- RC 2: Invalid context

  This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.

6.3 Query a General Entity

An Application may use this function to cause the Parser to return a ETY structure for a General Entity. To use this function, the Application must provide the name of the wanted entity.

The following fields in the PAC must be set by the Application prior to calling the service routine to query a general entity:

- PAC.DAD.P Pointer to an ASCIIZ string containing the entity name

- PAC.FUNC QGETY_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to the ETY structure for this Entity

Return Codes:

- RC 2: Invalid context

  This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.

- RC 4: Entity not in DTD (unknown)

6.4 Query the full list of General Entities

An Application may use this function to cause the Parser to return a list of all the General Entities defined in the DTD.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the list of general entities:

- PAC.FUNC QLGETY_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to an array of ETY structures
- PAC.DAD.LEN Number of ETY structures in the array

Return Codes:

- RC 2: Invalid context

  This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.

6.5 Query a Parameter Entity

An Application may use this function to cause the Parser to return a ETY structure for a Parameter Entity. To use this function, the Application must provide the name of the wanted entity.

The following fields in the PAC must be set by the Application prior to calling the service routine to query a parameter entity:

- PAC.DAD.P Pointer to an ASCIIZ string containing the entity name
- PAC.FUNC QPETY_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to the ETY structure for this Entity

Return Codes:

- RC 2: Invalid context This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.
- RC 4: Entity not in DTD (unknown)

6.6 Query the full list of Parameter Entities

An Application may use this function to cause the Parser to return a list of all the Parameter Entities defined in the DTD.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the list of parameter entities:

- PAC.FUNC QLPETY_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to an array of ETY structures
- PAC.DAD.LEN Number of ETY structures in the array

Return Codes:

- RC 2: Invalid context

  This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.

6.7 Query the Concrete Delimiter Strings

An Application may use this function to cause the Parser to return a table of strings, each string corresponding to a specific delimiter. The index of each specific delimiter within the table is listed in Figure 28.

```
#define COM_ix      0
#define CRO_ix      1
#define ERO_ix      2
#define ETAGO_ix    3
#define LIT_ix      4
#define LITA_ix     5
#define MDC_ix      6
#define MDO_ix      7
#define MINUS_ix    8
#define PERO_ix     9
#define PIC_ix     10
#define PIO_ix     11
#define REFC_ix    12
#define RNI_ix     13
#define STAGO_ix   14
#define TAGC_ix    15
#define VI_ix      16
#define MDO_DSO_ix 17
#define MSC_MDC_ix 18
```

Figure 28 DLMIX values. Index values for the delimiter table

NOTE: This is not an exhaustive list of all the delimiters, but most of the useful ones are here.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the delimiter strings:

- PAC.FUNC QDLM_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to a table of ASCIIZ strings

Return Codes:

- RC 2: Invalid context

This function can be called during the processing of any notification.

6.8 Query Information About Capacities

An Application may use this function to cause the Parser to return information about capacities.

The following fields in the PAC must be set by the Application prior to calling the service routine to query information about capacities:

- PAC.DAD.P Pointer to an array big enough for 3 pointers
- PAC.FUNC QCAPA_PSF function code

Upon return, PAC.DAD.P still points the array with the 3 pointers updated to respectively point to:

1. an array of unsigned long containing the count of each object,
2. an array of unsigned integer defining the value in capacity points for each object,
3. an array of unsigned long containing the maximum capacity points defined for each object.

Indexes to objects in all these arrays are defined in Figure 29. For instance "#define GRPCAP_ix 10" means GRPCAP is defined by entry #10 in any of these tables.

DTACAP and DTACHCAP don't exist in Figure 5 of ISO: those objects are associated to the number of DATATAG and Character of text in DATATAG. We thought it could be useful for an application to access this extra-information. The value in capacity points for those objects is 0 (since they are not really counted).

The entry at TOTALCAP_ix is relevant only for the array #3. For array #1, the entry at TOTALCAP_ix is reused to tell the number of Short References defined in the Concrete Syntax.

```
#define ATTCAP_ix     0
#define ATTCHCAP_ix   1
#define AVGRPCAP_ix   2
#define DTACAP_ix     3
#define DTACHCAP_ix   4
#define ELEMCAP_ix    5
#define ENTCAP_ix     6
#define ENTCHCAP_ix   7
#define EXGRPCAP_ix   8
#define EXNMCAP_ix    9
#define GRPCAP_ix    10
#define IDCAP_ix     11
#define IDREFCAP_ix  12
#define LKNMCAP_ix   13
#define LKSETCAP_ix  14
#define MAPCAP_ix    15
#define NOTCAP_ix    16
#define NOTCHCAP_ix  17
#define TOTALCAP_ix  18  /* only for the capamax table */
#define CAP_MAX_SIZE 19
```

Figure 29 CAP_IX values. Index values for the capacity tables

Return Codes:

- RC 2: Invalid context

  This function can be called during the processing of any notification.

6.9 Query Current Entity

An Application may use this function to cause the Parser to return a ETY structure for the current Entity. This service is primarily intended to find out what current entity is resumed after an entity is closed (ETYND_PFC).

The following fields in the PAC must be set by the Application prior to calling the service routine to query the current entity:

- PAC.FUNC QETY_PSF function code

Upon return, PAC.DAD.P points to the ETY structure for this Entity.

NOTES:

1. If there is no current entity, a NULL pointer will be in PAC.DAD.P (i.e. when the primary entity has been closed).
2. PI entities are not really opened by YASP, and therefore not shown by QETY_PSF, since they are not entered in the stack.

Return Codes:

- RC 2: Invalid context

  This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.

6.10 Query the ShortRef delimiter table

An Application may use this function to cause the Parser to return a table of strings, each string corresponding to a specific short reference delimiter.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the short reference delimiter strings:

- PAC.FUNC QSR_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to an array of SRTBEs. The SRTBE structure is described in Figure 30.
- PAC.DAD.LEN Number of SRTBE structures in the array

```
typedef struct {
   unsigned char *p;      /* Actual ShortRef string */
   unsigned int reserved; /* Reserved field */
}SRTBE;
```

Figure 30 SRTBE structure

Return Codes:

- RC 2: Invalid context

  This function can be called during the processing of any notification.

6.11 Query the declared quantities

An Application may use this function to cause the Parser to return information about quantities.

The following fields in the PAC must be set by the Application prior to calling the service routine to query information about quantities:

- PAC.FUNC QQTY_PSF function code

Upon return, PAC.DAD.P will point to an array of indexed integers: The index for each object is described in Figure 31.

```
#define ATTCNT_ix               0
#define ATTSPLEN_ix             1
#define BSEQLEN_ix              2
#define DTAGLEN_ix              3
#define DTEMPLEN_ix             4
#define ENTLVL_ix               5
#define GRPCNT_ix               6
#define GRPGTCNT_ix             7
#define GRPLVL_ix               8
#define LITLEN_ix               9
#define NAMELEN_ix             10
#define NORMSEP_ix             11
#define PILEN_ix               12
#define TAGLEN_ix              13
#define TAGLVL_ix              14
```

Figure 31 QTY_ix values. Index for quantity integer table

Return Codes:

- RC 2: Invalid context

  This function can be called during the processing of any notification.

6.12 Query the enabled features/options

An Application may use this function to cause the Parser to return the enabled features and options in the SGML declaration.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the enabled features and options:

- PAC.FUNC QOPT_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to an OPT structure.

The OPT structure is defined in Figure 32. The Feature Code bits flag is described in Figure 33. The Other bit flag is described in figure 34.

```
typedef struct {              /* Options */
   unsigned int ft;           /* Feature Code (Bits Flag) */
   unsigned int ot;           /* Others Bits Flag */
   unsigned int max_simple;
   unsigned int max_explicit;
   unsigned int max_concur;
   unsigned int max_subdoc;
   ADLEN    appinfo;          /* appinfo.p is also ASCIIZ */
}OPT;
```

Figure 32 OPT structure. Description of enabled features and options

```
#define FORMAL_FT         1   /* formal public identifiers */
#define CONCUR_FT         2   /* concurrent document instances */
#define EXPLICIT_FT       4   /* explicit link process */
#define IMPLICIT_FT       8   /* implicit link process */
#define SIMPLE_FT        16   /* simple link process */
#define SUBDOC_FT        32   /* nested subdocuments */
#define RANK_FT          64   /* omitted rank suffix minimization */
#define DATATAG_FT      128   /* datatag minimization */
#define OMITTAG_FT      256   /* omitted tag minimization */
#define SHORTTAG_FT     512   /* short tag minimization */
```

Figure 33 Feature Code bits flag

```
#define GENERAL_CASE_OT   1   /* upper case general */
#define ENTITY_CASE_OT    2   /* upper case entities */
#define INSTANCE_SCOPE_OT 4   /* is_DOCUMENT 0, is_INSTANCE 1 */
```

Figure 34 Other bit flag

Return Codes:

- RC 2: Invalid context

  This function can be called during the processing of any notification.

6.13 Query the function characters

An Application may use this function to cause the Parser to return the function characters defined in the SGML declaration.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the function characters:

- PAC.FUNC QFCT_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to an array of FNCR structures. The FNCR structure is defined in Figure 35.
- PAC.DAD.LEN Number of FNCR structures in the array

```
typedef struct {              /* to map : */
   union {                    /* a function name (ex: "TAB") */
     OFS ofs;                 /* offset... */
     unsigned char *p;        /* ...then relocated (when loaded) */
   }s;
   unsigned int ch_nb;        /* to a corresponding char number (ex: 9) */
   FCTCLS fctcls;             /* Function Class (ex: SEPCHAR_FCTCLS) */
}FNCR;
```

Figure 35 FNCR structure. Description of a function character

The Function Class values are defined in Figure 36.

```
typedef enum {
   NULL_FCTCLS = 0, /*  0 is Reserved */
   RE_FCTCLS,
   RS_FCTCLS,
   SPACE_FCTCLS,
   FUNCHAR_FCTCLS,
   MSICHAR_FCTCLS,
   MSOCHAR_FCTCLS,
   MSSCHAR_FCTCLS,
   SEPCHAR_FCTCLS
}FCTCLS;
```

Figure 36 FCTCLS values

Return Codes:

- RC 2: Invalid context

  This function can be called during the processing of any notification.

6.14 Query a Notation

An Application may use this function to cause the Parser to return a NCB structure for a Notation. To use this function, the Application must provide the name of the wanted notation.

The following fields in the PAC must be set by the Application prior to calling the service routine to query a notation:

- PAC.DAD.P Pointer to an ASCIIZ string containing the notation name
- PAC.FUNC QNOTA_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to the NCB structure for this Notation

Return Codes:

- RC 2: Invalid context This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.
- RC 4: Notation not in DTD (unknown)

6.15 Query the full list of Notations

An Application may use this function to cause the Parser to return a list of all the Notations defined in the DTD.

The following fields in the PAC must be set by the Application prior to calling the service routine to query the list of general entities:

- PAC.FUNC QLNOTA_PSF function code

Upon return, the following field is set in the PAC:

- PAC.DAD.P Pointer to an array of NCB structures
- PAC.DAD.LEN Number of NCB structures in the array

Return Codes:

- RC 2: Invalid context

  This function can only be called when the parsing of the prolog has finished, i.e., when the PROND_PFC notification has occurred.