Field Device Web Server Release 1.0 Software User Manual

ALS 53424 a–en

First issue: 02–2000

Meaning of terms that may be used in this document / Notice to readers

WARNING

Warning notices are used to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury exist or may be associated with use of a particular equipment.

In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.

Caution

Caution notices are used where there is a risk of damage to equipment for example.

Note

Notes merely call attention to information that is especially significant to understanding and operating the equipment.

Index letter	Date	Nature of revision

Revisions

1. PURPOSE OF MANUAL AND DOCUMENTED VERSION

This manual describes the collection of software packages called in the sequel FieldDeviceWebServer<>. Its main objective is to facilitate the design and implementation of software applications called ***Embedded Web Sites*** or interchangeably, ***Embedded Web Servers.*** The document contains the detailed description of the functions contained in the software packages and is considered as the reference manual for designers of the embedded servers using this technology as support for their projects.

2. CONTENT OF THIS MANUAL

Chapter 1 – Presentation: this chapter presents the background of the FieldWebServer as well as its architecture.

Chapter 2 – General information: this chapter introduces basic information about the utilization context and the specific package (Cocktail). It also deals with the start/stop procedures.

Chapter 3 – Use: this chapter presents, for the four packages, the functions and procedures of each module.

3. RELATED PUBLICATIONS

- INTERNET COMMUNICATIONS STEP 1: Software Development Plan – Y3–30 A418982 A. **[1].**
- INTERNET COMMUNICATIONS STEP 1: Internet Techniques in Distributed Control Systems Tutorial – Y3–30 A419463–1. **[2].**
- INTERNET COMMUNICATIONS STEP 1: System Specifications and Design Document – Y3–30 A419481–A. **[3].**
- SOCKAPI SOCKETS Software Specification Document – Y3–30 A419465–B. **[4].**
- J. Grosch – Generators for High Speed Front Ends, Report N°.11 ,CoCoLab –Datenverarbeitung. **[5].**
- J. Grosch Rex – A Scanner Generator, Report N°.5 ,CoCoLab –Datenverarbeitung. **[6].**
- J. Grosch Lark – An LR(1) Parser Generator with Backtracking, Report N°.32, CoCoLab –Datenverarbeitung. **[7].**
- J. Grosch Puma – A Generator for the Transformation of Attributed Trees, Report N°.26 ,CoCoLab –Datenverarbeitung. **[8].**
- J. Grosch Ast – A Generator for Abstract Syntax Trees, Report N°.15 ,CoCoLab –Datenverarbeitung. **[9].**
- T.Berners–Lee, R.Fielding, H.Frystyk: Hypertext Transfer Protocol –HTTP /1.0 , Network Working Group, RFC 1945. **[10].**

4. WE WELCOME YOUR COMMENTS AND SUGGESTIONS

ALSTOM strives to produce quality technical documentation. Please take the time to fill in and return the ”Reader’s Comments” page if you have any remarks or suggestions.

ALS 53424 a–en ***Field Device Web Server Release 1.0 Software User Manual***

Your main job is:

☐	System designer	☐	Programmer
☐	Distributor	☐	Maintenance
☐	System integrator	☐	Operator
☐	Installer	☐	Other (specify below)

If you would like a personal reply, please fill in your name and address below:

COMPANY: . . . . NAME: . . . .

ADDRESS: . . . .

. . . . COUNTRY: . . . .

Send this form directly to your ALSTOM sales representative or to this address:

ALSTOM Technology
Technical Documentation Department (TDD)
5 avenue Newton BP 215
92142 Clamart Cedex
France
Fax: +33 (0)1 46 29 12 44

All comments will be considered by qualified personnel.

REMARKS

Continue on back if necessary.

Reader's comments

Contents

Contents

Contents

Figures

Tables

Chapter 1 Presentation

This manual describes the collection of software packages called in the sequel Field Device Web Server. The rationale for this product is to support the technology which enhances control system architectures, and especially fieldbus–based parts of these architectures, with the potential for INTERNET–compatible processing.

The use of this technology allows control systems to take advantage of the INTERNET type client–server architecture where elements of the user interface to applications placed in field devices, including their look–and–feel part, are provided by the server.

It is worth stressing that one of the most important features of the technology developed and supported by the software described below is that it should provide components which **enhance** rather than **replace** existing applications hosted by a control device. The principal mission of a device in which the embedded INTERNET components are to be installed is by no means changed.

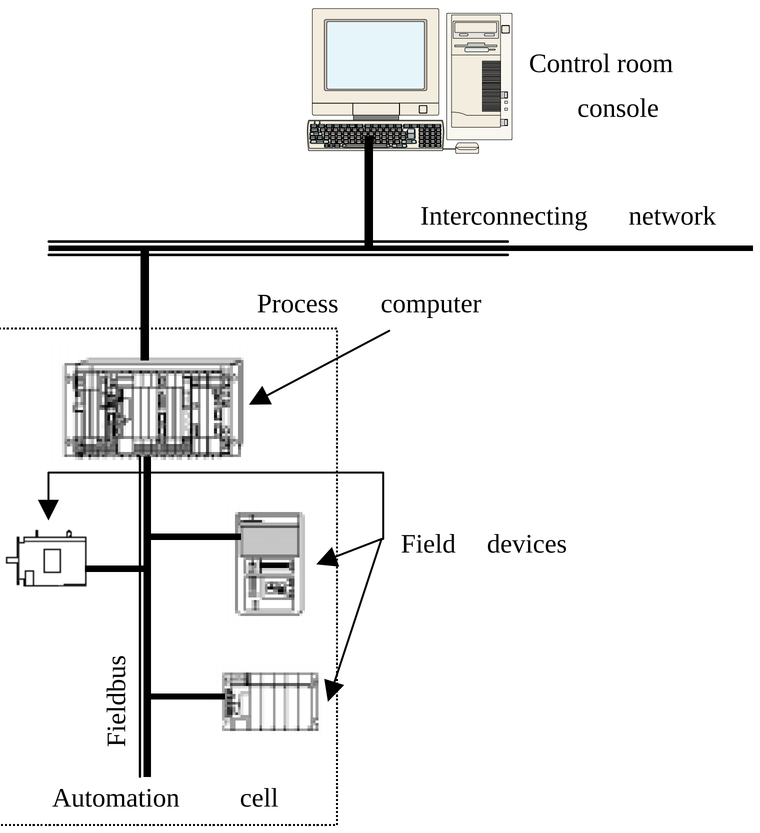


Figure 1.1 – Place of field devices in automation system

To sketch the impact of the technology it is important to identify the place of a field device in a typical architecture of a control system, as shown in Figure 1.1. The field device is part of an *automation cell* – a collection of co–operating instruments which provide a well defined automation function. These devices are of different levels of complexity – from simple sensors with extremely limited functions to process computers equipped with powerful processors and large memory banks containing large quantities of embedded software.

Automation cells of a control system co–operate in order to implement a coherent control application. The co–operation is possible thanks to information exchanges via a higher level network which forms the system backbone. This backbone links the automation cells with the control room supervisory computers which implement the interface between the system and human operators. In most cases control applications are arranged in two sets of functions: automation functions implemented in automation cells and supervisory functions implemented in control room computers.

Information exchange between the two parts is based almost exclusively on the client–server paradigm. Data streams exchanged between control room client software and automation cell servers transit via the backbone network of the system, which in most cases is based on the Ethernet link layer and supports information transport via the TCP/UDP/IP protocol stack. Application data exchange is implemented by the variety of higher level protocols, mostly proprietary and sometimes standardised (MMS, FTP).

Application protocols influence the structure of client and server software which varies greatly from one system to another. Non standard protocols result in non standard solutions in terms of services provided, flexibility, performance and complexity. This lack of compatibility strongly reduces the migration of software components from one application to another and virtually excludes the portability of control applications from one system to another.

The idea which initiated launching of the development process of the Field Device Web Server had its origins in the analysis of INTERNET structure. This world–wide communication support is one example of a successful application of the interoperability principle applied to diverse software products. The interoperability of INTERNET products (clients and servers) is based on universally accepted standards which are:

- the TCP/IP protocol for reliable data transport,
- the HTTP protocol for application information exchange,
- the HTML format for information structuring and presentation.

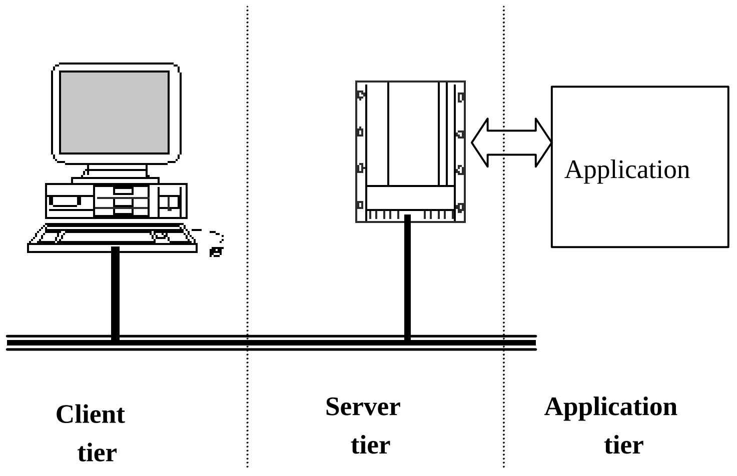


Figure 1.2 – Three–tier architecture of INTERNET client–server application

INTERNET–distributed applications are based on the principle of ***a three–tier*** architecture Figure 1.2 which makes use of universal client and server frameworks independent of the nature of data processed, on condition that the data exchanged over the network are transported via the HTTP protocol and are structured according to the HTML format. The three–tier architecture standardises basic services of the client and server parts of the application. In this configuration the client part is totally independent of the application (so called ***thin client***). The burden of application personalisation is placed on the server side, which is interfaced directly with the embedded application software. This configuration is at the origin of the internal architecture of the server tier which contains three functionally separate parts:

- generic modules of the server which support HTTP protocol operation, totally independent of the user application,
- user–provided modules implementing interaction mechanisms between the human operator and the embedded application software. These modules contain passive data such as embedded HTML pages, embedded images and embedded Java applets as well as active data such as routines generating dynamic pages or routines embedding process data into served HTML pages,
- application wrapper modules which adapt the API exposed by the embedded application to the needs of user–provided modules.

This architecture simplifies the development process of newly created applications. Only user–provided modules are to be developed for each new project. Generic modules are to be totally reused for a wide range of environments and for different types of applications. Wrapper modules are to be developed for each type of hosting device and reused for different applications placed in the given type of the device.

1. GENERAL DESCRIPTION OF DEVELOPED SOFTWARE

The introductory explanation from the previous section shows that there are two conditions for setting up INTERNET technology inside the automation cells.

- enhancement of fieldbus protocol stack with the TCP/IP layer; the cell front–end (process) computer must in this case play the role of IP router,
- placing the server part of the distributed application in a device within the cell; this device should be equipped with sufficient processing power and should have enough memory to host the server modules; in many cases the server is placed in the cell front–end computer.

The software structure of the device hosting the server will thus be composed of two functionally specialised parts:

- the essential application doing the real job of process control; this application is in principle independent of the INTERNET,
- the server tier from Figure 1.2 adapting the interaction mode between the generic thin client and the essential application.

Since elements of automation cells do not dispose of large computing resource reserves it is necessary to develop the components allowing the user to construct with ease small, portable and modular server tiers. In most cases every server tier is composed of three basic elements:

- generic server body – principal active component which loops heaping to the incoming service requests and processes them; request processing consists in parsing the PDU syntax, recovering environment variables in order to support CGI operations, identifying requested resources together with the operations to be applied to them: the generic server body is in principle independent of the applications in which it is incorporated; its basic elements are server engine, request parser, response composer and persistence module,
- server object repository – active component implementing the logistics of server object management; this component structures the collection of objects which together make up the application, and is in charge of assembling the elements which incorporate information generated by the application,
- collection of application specific components – elements which implement both the look–and–feel part of the application (HTML pages, compressed images, Java applets, ActiveX controls) and its dynamics (CGI scripts, servlets); it is important to state that these components do not do the real job of the application; their role consists in conveying data between the client part and the essential application; most naturally these elements are totally application dependent.

The analysis of the structure of an embedded server highlights yet another building block of the architecture – the application wrapper. This block is very often introduced into the device structure for reasons of convenience. Its role consists in adapting the functional interface of the basic application to the needs of the page composition module. The structure of this block is totally dependent on the basic application and on the design of application specific components of the server. The construction of this block is not supported by the modules of the Field Device Web Server and for this reason it is left outside of the server tier structure.

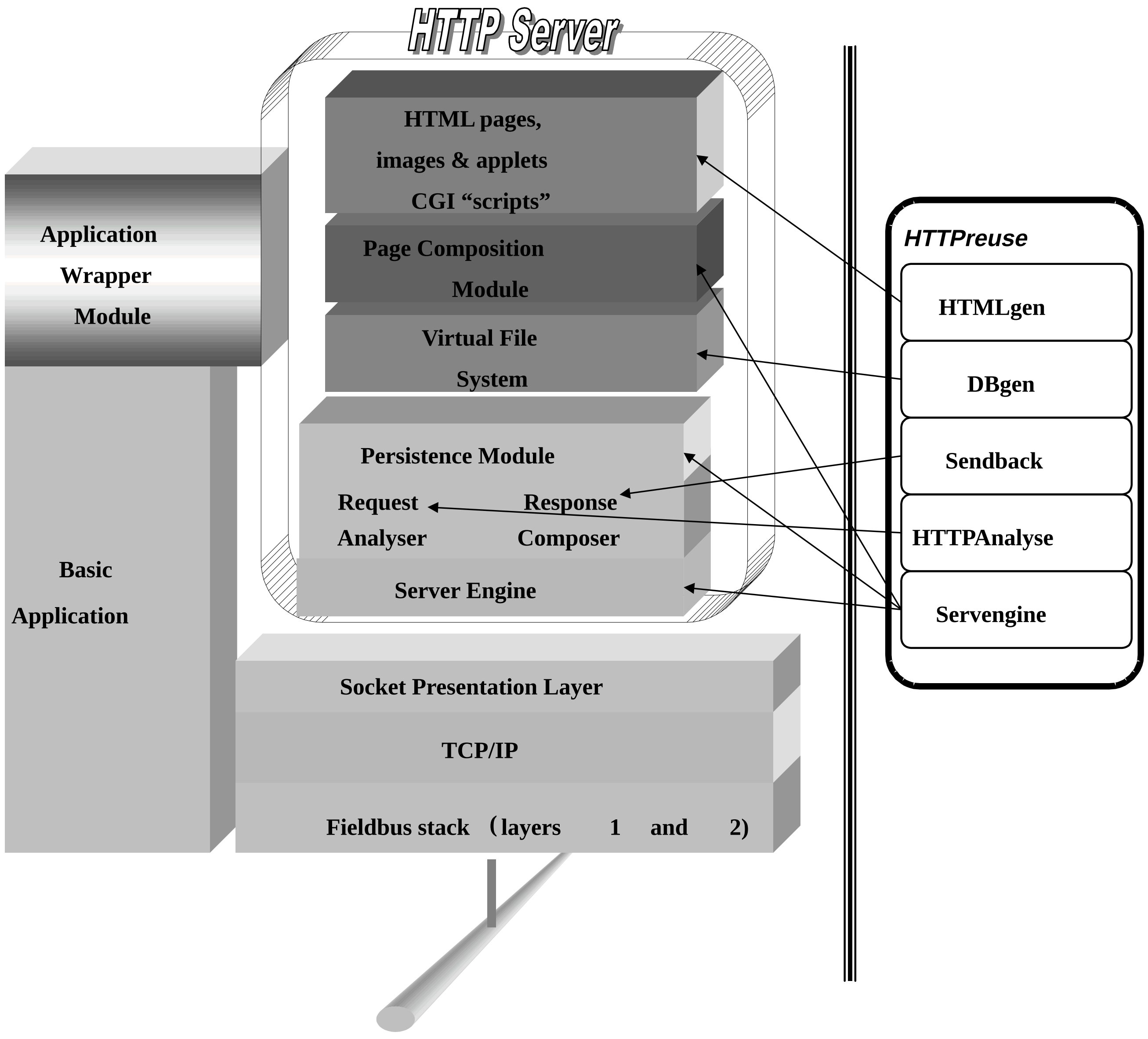


Figure 1.3 – Architecture of INTERNET–based application in an Alspa F8000 equipment

Taking into account these considerations, the entire INTERNET–based server architecture, in the context of a control device, can be represented by the schematic in Figure 1.3.

The left part of the schematic above shows the software architecture at run time, which highlights mutual relationships between all building block instances. The right part of the schematic shows the organisation of the Field Device Web Server module library, called in the context of this project ***HTTPreuse*** at development time. As shown in Figure 1.3, it is composed of five interrelated software packages, set up in order to support configuration management:

- **Servengine** package: this package groups the modules which provide the functions of server engine operation, network adaptation and persistence of request data,
- **HTTP Analysis** package: this package implements the functions of request analysis and CGI environment building,
- **DBgen** package: this package provides the elements which support the implementation of the embedded equivalent of a disk file system,
- **HTMLgen** package: this package provides the elements which support the implementation of embedded HTML pages,
- **Sendback** package: implements basic response composition functions.

The following figure shows the mutual relationships between the packages:

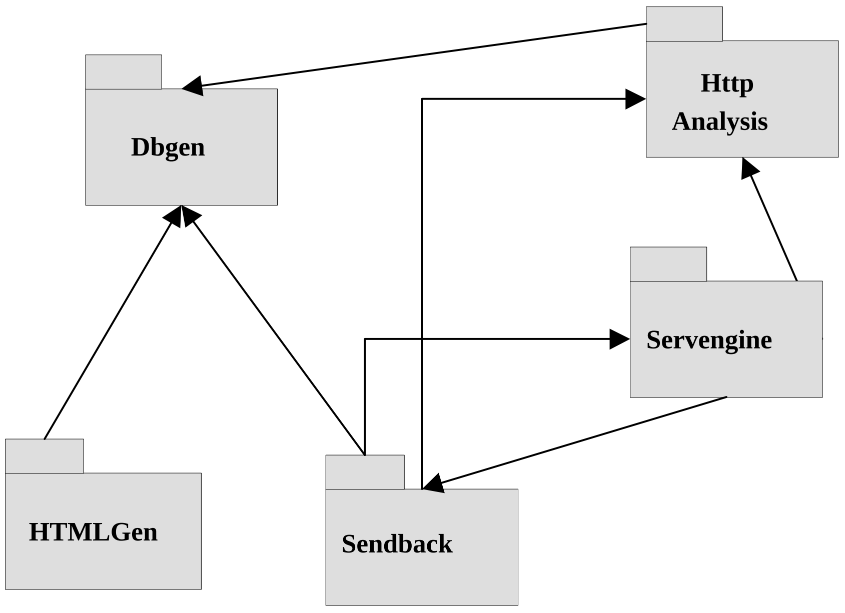


Figure 1.4 – General architecture of Field Device Web Server software

Chapter 2 General information

1. UTILISATION CONTEXT

The software described in this manual is in principle independent of the implementation platform i.e. the processor type, operating system and network stack. A few implementation details can be adapted to almost any environment, provided that the environment supports ANSI C run –time routines. Clearly, you need to use the development tools (compiler, linker, loader) to install the generated code in its environment. The unique requirement imposed on the hardware platform consists in the availability of free space in the processor memory and in access to a version of the socket presentation interface of the TCP/IP network. In principle, the software is independent of the type of socket implementation and operating system.

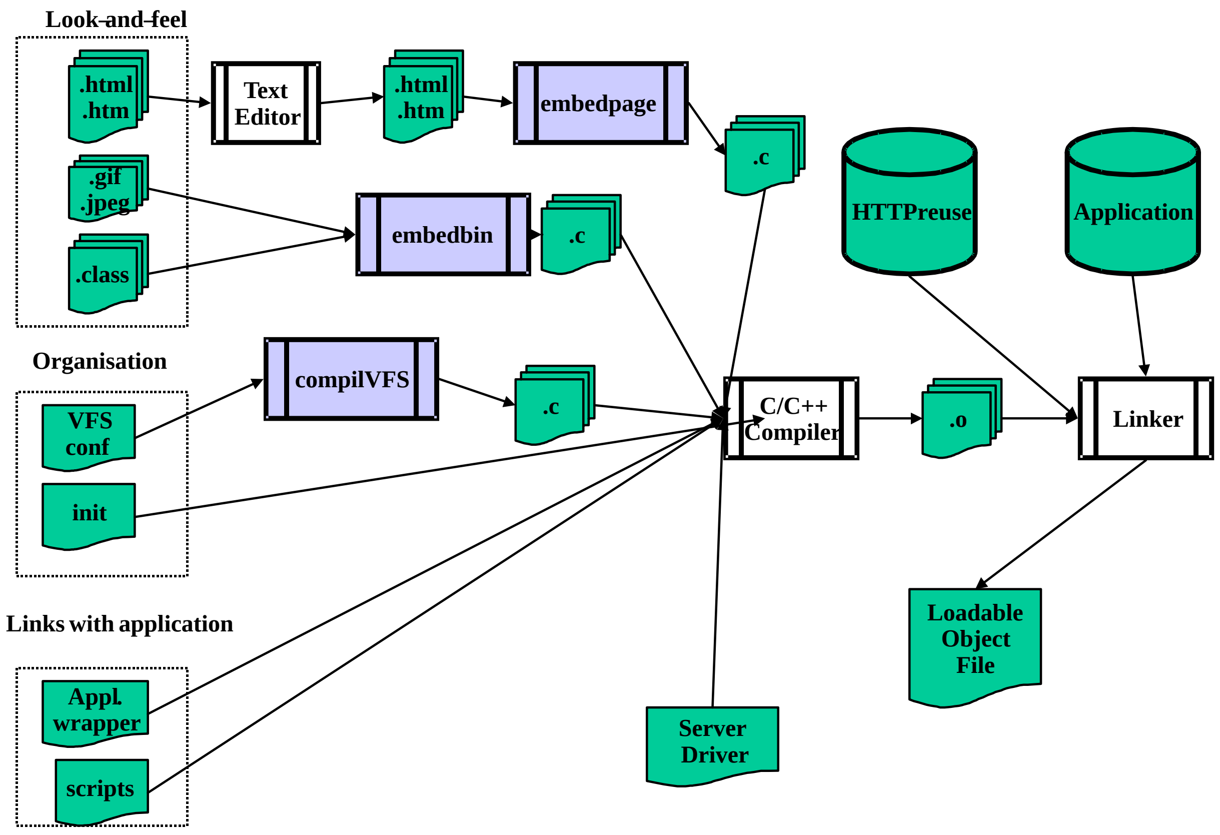


Figure 2.1 – Development process for an embedded server application

The generic modules provided by the HTTPreuse library are the most complex part of the server tier application. Despite this fact the application developer tends not to see them in the process of application development. His attention is naturally focused on five aspects of the application which are:

- application start–up: a set of operations activated before the server loop is bootstrapped; these operations initialise other application components; their implementation is done manually by assembly of parts from HTTPreuse packages,
- embedded site organisation; this part of the application concerns the design of the Virtual File System (VFS) of the server tier; the VFS holds the references of all objects which are part of the site and provides the search mechanism for server objects: pages, applets, images and CGI scripts; the application developer constructs this part of the embedded site by using the routines from the package described in chapter 3 § 1; it is worth noting that non trivial sites can be tedious and complicated but can be easily mechanised by using a configuration tool (see Appendix B),
- embedded look–and–feel objects which are data structures representing embedded passive objects (page frames, applets, images); these data are represented as octet arrays residing in memory regions accessible via VFS ; in the normal development process these objects are designed and implemented by tools adapted to the object nature (HTML editors, image editors, JAVA development tools), a necessary step concerning the transformation of standard formats of their representations (ASCII files, GIF/JPEG files, byte–code) to byte arrays loadable to device memories is to be supported by appropriate tools (see Appendix B),
- page composing routines which allow the user to merge passive objects (page frames) with application data in order to form a complete HTML pages which incorporate application status; these routines are to be programmed manually or are to be generated from a user friendly notation (see Appendix C),
- routines representing active server objects (CGI scripts, dynamic pages) executed on requests received from the client; these routines serve to integrate application data to server pages; the routines reuse generic functions provided by HTTP reuse; their design is so dependent on the application that only manual development process is possible.

The development process of the specific application is presented in Figure 2.1. Relationships between application components are presented in Figure 2.2.

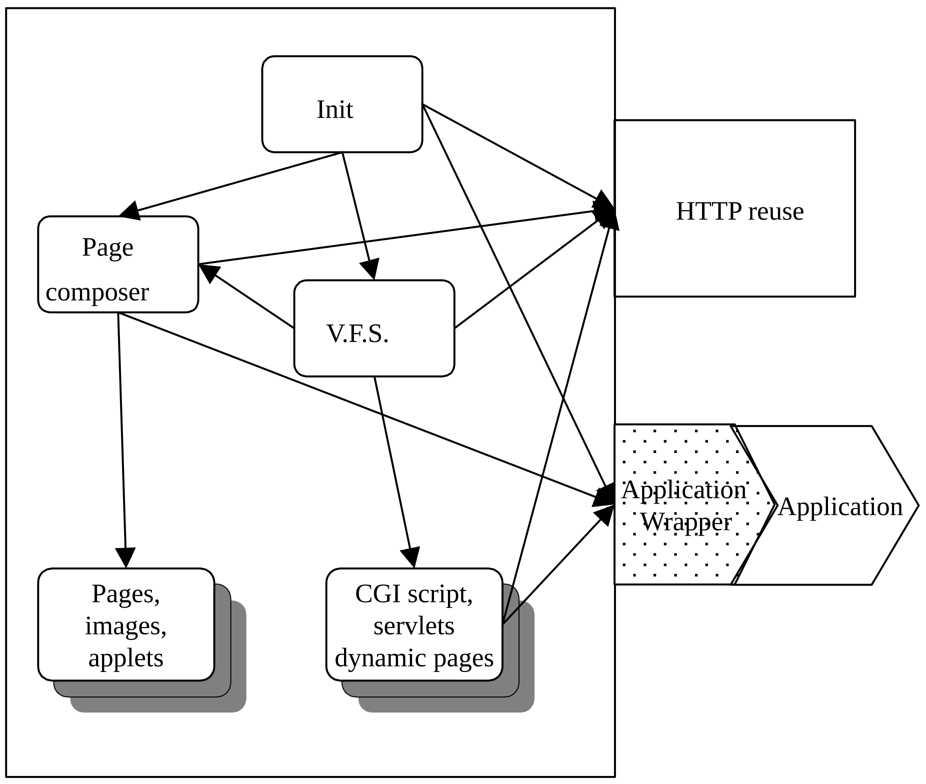


Figure 2.2 – Detailed view of embedded software application

2. RÔLE OF COCKTAIL TOOLBOX

One of the important functions of the developed system consists in analysing the requests incoming from the network. The software package which implements this function operates as an analyser of a stream of ASCII encoded characters. This makes its design very similar to the design of front end modules of programming language compilers. Such modules are never programmed manually but are generated from the high level specification of the syntax of the character stream. To generate the analysers one uses the software known under the generic name of compiler compilers. The most eminent (but not the most powerful) example of such a software is the pair of UNIX programs LEX and YACC.

The technique of implementation of request analysers used in this project is based on the principle of automatic generation of the analysers. The tools used in the project belong to the COCKTAIL **[5,6,7,8,9]** package – which is one of the most powerful tool sets known in the area of compiler compilers.

It is composed of a coherent set of processors which support the following operations:

- **rex** – lexical analyser counterpart of lex; generates scanners four times smaller and four times faster,
- **lark** – counterpart of yacc; among many features which make it more user friendly than yacc, this processor generates parsers which are twice as fast as parsers generated by its well–known predecessor for approximately the same size of code,
- **ag** – evaluator of attributed grammars not used here; does not have any counterpart among freeware programs,
- **ast** – abstract syntax tree generator used here to generate the programs in the modules **`data_base_structure`** **and** **`html_page_elements`**,
- **puma** – supports tree evaluation and transformation; used in modules **`data_base_processing,`** **`gen_tree_module`** and **`html_conv_module.`**

The use of these tools considerably increases the speed and the quality of the development process. This can be explained by the size of the source files which directly influences the ease of code maintenance.

Another advantage of the COCKTAIL package is the existence of a library of reusable software components which are used not only by the routines generated by the above cited tools but can also provide services to hand written programs.

3. START–UP/STOP FUNCTIONS

3.1. Start–up procedure

The produced software is a collection of software routines to be used by an application designer. As such it does not have a well defined and unique start–up procedure per se. It provides only the elements to be used during the start–up phase of the application developed with the use of this software. The recommended calling sequence for initialisation of an embedded server application is as follows:

1. Initialise network interface and system support such as memory management operations.
2. Initialise VFS.
3. Initialise persistence component.
4. Initialise user application interface.
5. Bootstrap the server execution loop.

The server execution loop itself contains an initialisation sequence which is either provided in the standard version of the loop or should be developed according to the design of the customised version of the server engine.

3.2. Stop procedure

For the reasons evoked in the previous section this software does not provide an explicit stopping procedure. The other reason for this situation is the fact that the possibility of stopping an embedded server is not a commonly required feature. However such a possibility exists, since the module providing the server engine routines gives access to the variable which controls the server loop (see chapter 3 § 2.1). This design feature provides the possibility of server stopping and restarting.

Chapter 3 Use

1. PACKAGE *VFS GENERATION*

The package regroups the modules which provide the services linked to the generation and handling of the Virtual File System. The package is composed of three modules:

- module ***commondef*** widely used all over the software, it defines the primitive data type **`pchar`** widely used in the whole Field Device Web Server software;
- module ***data_base_struct*** defining the fundamental data type **`tdata_base_struct`** on which the construction of the VFS is based;
- module ***data_base_processing*** providing the routines which are used in the processing of the elements of VFS.

The relationships between the modules of the package are shown in the diagram in Figure 3.1.

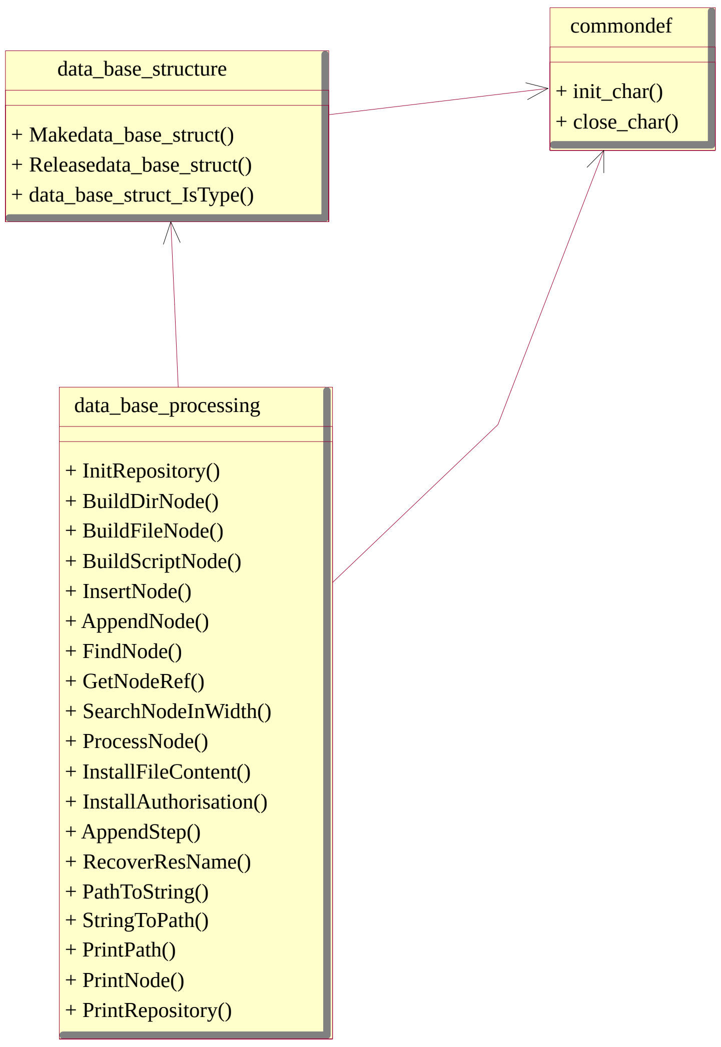


Figure 3.1 – Module relationship within the package *VFS Generation*

1.1. Module *Commondef*

This module groups the data type definitions necessary for other modules from the package in which it is implemented but also for other packages. It provides directly one data type: `pchar` which is the pointer to a heap allocated character string. This type has two operations directly associated with it: `init_char`, which is the string creator and initialiser and `close_char` which is the string destructor. Two other types: `uchar` and `ushort` are defined by renaming. All other type descriptions are imported from standard definitions via appropriate ***#include*** **<...>** clauses. As far as this operation is concerned the module plays the role of the adapter to different compiler environments.

1.1.1. Function *char * init_char(const char*)*

1.1.1.1. Description

This routine implements the creation of a character string. If the input parameter references an existing string of non zero length, the routine attempts to allocate on the system heap a memory zone which has the length of the input string length +1. If the allocation succeeds, the input string is copied into the zone. If the input parameter is NULL or if the referenced string has zero length or if the allocation fails, the routine returns NULL.

1.1.1.2. Inputs

The unique input parameter is a pointer referencing an existing non empty character string.

1.1.1.3. Outputs

The routine returns a pointer to the newly created string in the system heap. The string is the clone of the input string and has the termination character.

1.1.1.4. Fault messages

The routine returns NULL pointer if the input string is NULL or has zero length or if the attempt to allocate the needed heap frame fails.

1.1.2. Procedure *void close_char(char*)*

1.1.2.1. Description

This routine implements the destruction of the character string created by init_char. The heap region allocated to the string is freed but not set to zero.

1.1.2.2. Inputs

Pointer referencing the string to be destroyed.

1.1.2.3. Outputs

None.

1.1.2.4. Fault messages

None.

1.2. Module *data_base_structure*

This module defines types and routines on which the Virtual File System is constructed. The most important item defined by the module is the data type **`tdata_base_structure.`** Although the file is written in ANSI C programming language, its small part is directly coded in C. The vast majority of the code, including the data type definition is automatically generated by **ast**, one of the processors which makes up part of the COCKTAIL toolbox (see chapter 2 § 2). The original specification of the module is done using the **ast** syntax and can be found in the file **`data_base.ast`**.

Apart from the data type definition the module provides the implementation of three auxiliary functions, as described below.

1.2.1. Definition of `tdata_base_structure` data type

The definition of **`tdata_base_structure`** follows the regular scheme, as below:

typedef union {
yobj_type_name_1 obj_type_name_1 ;
yobj_type_name_2 obj_type_name_2 ;

yobj_type_name_n obj_type_name_n ;} * tdata_base_structure;

The type is then a pointer to a union which groups the structures. Each variant of the union designates the type of object belonging to the VFS. These types have a very consistent construction as shown in the schematic below:

typedef dll_obj_desc_tKind unsigned char;

typedef { dll_obj_desc_tKind Kind ;

dll_obj_desc_tNodeHead yyHead ;

●

object attribute fields

●

●

} yobject name ;

The first field of each object is the tag **`Kind`** which identifies the object class. The second field of the record is used by the VFS internal modules to manage the pool of objects.

The rest of each structure is composed of the series of object attribute definitions. In principle, the user should not modify these attributes manually and should rely on the set of routines furnished by this module as well as by the module ***data_base_processing***.

The type of the first field of the structure is the sub type of **`unsigned char`** and takes on the values from 1 to 9. The type values are redefined as a set of symbolic constants according to the following scheme:

Value	Symbolic name attributed	Kind of object designated
1	krepository	abstract object class from which are inherited all others except *path* and *access_list*
2	kroot	root of a Virtual File System tree
3	knode	abstract class from which *embedded_directory* and *embedded_active_node are inherited*
3	kembedded_directory	intermediate node in a VFS, it can group other nodes
5	kembedded_active_node	abstract class of leaf nodes of VFS; *embedded_file* and *embedded_script* are inherited from it
6	kembedded_file	leaf of the VFS which represents a file containing an html page, an image or an applet
7	kembedded_script	leaf of the VFS designating an executable routine
8	kpath	link to a node in the VFS
9	kaccess_list	list of access control records composed of pairs username:password

Table 3.1 – Type values and their mean

In the current version of VFS there are nine types of objects divided into three groups:

- three VFS abstract objects: **`repository, node`** and **`embedded_active_node`**; these objects are used very rarely since they represent the common properties of other objects,
- three classes of VFS concrete objects: **`embedded_directory,`** **`embedded_file`** and **`embedded_script`**; these objects represent the concrete elements of the VFS system corresponding to directories and files of a regular disk–oriented mass memory system,
- two classes of auxiliary objects: **`path`** and **`access_list`**; these objects represent respectively a link to a VFS node and the list of access right records (pairs of username + password) used to restrain the navigation in the VFS.

Each object features a number of attributes. The non abstract object attributes are specified and described in the following table:

Object name	Attribute name	Attribute type	Description
root	name	pchar	pointer to a character string which stores the name of the whole VFS tree
	member_list	tdata_base_struct	pointer to a list of sub–trees of VFS tree; the sub–trees can be either leaves (embedded files or embedded scripts) or sub–tree roots (embedded directories)
	access_list	tdata_base_struct	pointer to a list of access records each storing a pair of character strings username:password; this feature provides the access control for the whole VFS tree
	default_obj_name	pchar	character string which contains the name of the default file served by the server when there is no resource designated by the HTTP service request (URL is limited to the server TCP/IP address only)
	next	tdata_base_struct	reference of another tree of the VFS residing in the same server
embedded_directory	name	pchar	pointer to the character string storing the name of the directory
	member_list	tdata_base_struct	pointer to a list of sub–trees belonging to the directory; the sub–trees can be either leaves (embedded files or embedded scripts) or sub–tree roots (embedded directories of lower level)
	access_list	tdata_base_struct	pointer to a list of access records each storing a pair of character strings username:password; this feature provides the access control for the directory
	next	tdata_base_struct	reference of an adjacent node (file or directory) at the same level of the VFS
embedded_file	name	pchar	pointer to the character string storing the name of the file
	file_id	unsigned short	unique file serial number attributed to all VFS leaves
	nature	unsigned short	code describing the file formats; can take on the values: c_text(1), c_gif(2), c_jpeg(3), c_java(5), c_script(6), (c_dir(4) corresponds to the directory file)
	contents	pchar	pointer to a chain of bytes which stores the contents of a page, an image or an applet
	size	unsigned short	length of the file expressed in number of bytes; if this value is equal to 0 this means that the file contains a zero–terminated character chain
	next	tdata_base_struct	reference of an adjacent node (file or directory) at the same level of the VFS
embedded_script	name	pchar	pointer to the character string storing the name of the file
	file_id	unsigned short	file serial number attributed to all VFS leaves
	nature	unsigned short	code describing the file formats; can take the values: c_text(1), c_gif(2), c_jpeg(3), c_java(5), c_script(6), (c_dir(4) corresponds to the directory file)
	script_exec	tscript	pointer to a sub routine which corresponds to the script
	next	tdata_base_struct	reference to an adjacent node (file or directory) at the same level of the VFS

Table 3.2 – Non Abstract object attributes

Object name	Attribute name	Attribute type	Description
path	step	pchar	pointer to the character string storing the name of the step in the path
	next	tdata_base_struct	reference to the next path record; if this field is positioned to NULL the current step is the last
access_list	user	pchar	pointer to the character string storing the name of the user
	password	pchar	pointer to the character string storing the name of the password corresponding to the user name
	next	tdata_base_struct	reference to the next access record; if this field is positioned to NULL this means the end of list

Table 3.2 – Non Abstract object attributes (continued)

As it was stated above, the user of the Field Device Web Server library is not supposed to manipulate directly the data type specified above. The majority of operations are encapsulated by the programs grouped in the modules of this package and in the package ***data_base_processing***.

1.2.2. Function *tdata_base_struct Makedata_base–struct(unsigned char Kind)*

1.2.2.1. Description

This routine creates in the system heap an object of the kind specified by the unique parameter and returns the reference to this object.

1.2.2.2. Inputs

The input parameter specifies the kind of object to be created; its value should correspond to one of the constants specified in Table 3.1.

1.2.2.3. Outputs

The returned value is a reference of the newly created object (pointer to the object) if the creation succeeds.

1.2.2.4. Fault messages

In case of failure the procedure returns the NULL pointer. No explicit textual message is generated.

1.2.3. Procedure *void Releasedata_base–struct(tdata_base_struct obj)*

1.2.3.1. Description

This routine destroys the object referenced by the input parameter. The heap area occupied by the object is returned to the system.

1.2.3.1.1. Inputs

The input parameter references the object to be destroyed.

1.2.3.1.2. Outputs

None.

1.2.3.1.3. Fault messages

None.

1.2.4. Function *rbool data_base_struct_IsType(tdata_base_struct obj, unsigned char Kind)*

1.2.4.1. Description

This routine checks whether the object referenced by its first parameter is of the type designated by the second parameter. The object is considered to be of the given type in two cases:

- it belongs to a sub type of the specified type; *example: root* is of type *krepository,*
- it is strictly of the specified type; *example: embedded_file* is of type *kembedded_file.*

1.2.4.2. Inputs

The first input parameter references the object, the second provides the type to be checked with.

1.2.4.3. Outputs

The function returns:

1	when the object is of the specified type,
0	when it is not.

1.2.4.4. Fault messages

None.

1.3. Module *data_base_processing*

This module is composed of twenty one procedures which allow the user to create and handle the objects of the Virtual File System without requiring deep insight into its data structures. Although the procedures are compiled from the text in ANSI C programming language, none of them is directly coded in C. The code is automatically generated by **puma**, one of the processors which is part of the Cocktail toolboxs. (see chapter 2 § 2). The original specification of the module is done using the **puma** syntax and can be found in the file `data_base.puma`. Despite this programming technique the user interface to the module observes the ANSI C syntax.

Within these nineteen routines, fourteen serve for run time VFS handling while five of them are used mostly in debug or test mode. Among sixteen directly usable procedures, eleven manipulate Virtual File System nodes (files, scripts, directories, repositories), one handles authorisation installation and two process file reference elements (access path and extension string).

Apart from the routines mentioned above the module exports one data type which describes the type used by one of the exported routines : *ProcessNode*. The detailed description of this type can be found in the section describing the routine (Process Node § 1.3.10.).

1.3.1. Function *tdata_base_struct InitRepository(tdata_base_struct node, pchar default, pchar name*)

1.3.1.1. Description

This routine creates the root of the new Virtual System Tree in the system heap. It is usually the first operation in the process of building the server’s repository.

1.3.1.2. Inputs

The routine has three inputs:

- **node** – reference of an existing node which becomes the first internal node in the repository; if this parameter has NULL value the newly created tree will be reduced to the root node only;
- **default** – pointer to a character string which contains the name of the default file, served when the root of the VFS tree is invoked by the http request; the string should represent the name of an existing file;
- **name** – pointer to the string which contains the name of the newly created tree; this string van be empty but some routines of this module write return fault messages.

1.3.1.3. Outputs

The routine returns the pointer to the newly created root node of the VFS tree.

1.3.1.4. Fault messages

If the node creation fails the routine returns NULL.

1.3.2. Function *tdata_base_struct BuildDirNode(pchar name)*

1.3.2.1. Description

This routine creates in the system heap a new node of the type **`embedded_directory`**.

1.3.2.2. Inputs

The only input of the routines is a pointer to a character string which contains the directory's name.

1.3.2.3. Outputs

The routine returns the pointer to the newly created **`embedded_directory`** object.

1.3.2.4. Fault messages

If the node creation fails the routine returns NULL.

1.3.3. Function *tdata_base_struct BuildFileNode(pchar name, pchar contents, unsigned short size, unsigned short fid, unsigned short nature)*

1.3.3.1. Description

This routine creates in the system heap a new node of the type **`embedded_file`**.

1.3.3.2. Inputs

The routine has the following inputs:

- **name** – pointer to a character string which identifies the file within the VFS tree;
- **contents** – pointer to a contiguous memory area (byte string) containing the file contents;
- **size** – number of bytes contained in the area; if this parameter is equal to 0 the file contents is considered to be a zero–terminated character string;
- **fid** – numeric file identifier – support for fast file referencing;
- **nature** – code of file contents – it can take on the following values:

  c_unknown(0), c_text(1), c_gif(2), c_jpeg(3), c_script(6), c_java(5) and c_dir(4).

1.3.3.3. Outputs

The routine returns the pointer to the newly created embedded_file.

1.3.3.4. Fault messages

When the node creation fails the routine return a NULL pointer.

1.3.4. Function *tdata_base_struct BuildScriptNode(pchar name,tscript exec, unsigned short fid)*

1.3.4.1. Description

This routine creates in the system heap a new node of the type `embedded_script.`

1.3.4.2. Inputs

- **name** – pointer to a character string which identifies the file within the VFS tree;
- **exec** – pointer to a routine which is executed when the node is referenced in a http request this pointer has the type `tscript` exported by the data_base_struct module; the type definition is:

  typedef void(*tscript)(int,…)

- **fid** – numeric file identifier – support of the fast means of file referencing.

1.3.4.3. Outputs

The routine returns the pointer to the newly created embedded script.

1.3.4.4. Fault messages

When the node creation fails the routine return a NULL pointer.

1.3.5. Procedure *void InsertNode(tdata_base_struct dir, tdata_base_struct node)*

1.3.5.1. Description

This routine adds the VFS node, referenced by its second parameter, to the directory referenced by its first parameter.

1.3.5.2. Inputs

The two procedure inputs are:

- **dir** – pointer to an existing directory node; if this pointer is equal to NULL no operation is performed;
- **node** – pointer referencing an existing VFS node which can be of an `embedded_file`, an `embedded_script` or an `embedded_directory` kind; if the pointer does not point to a node of an appropriate kind, no operation is performed.

1.3.5.3. Outputs

None.

1.3.5.4. Fault messages

None.

1.3.6. Procedure *void AppendNode(tdata_base_struct root, tdata_base_struct node)*

1.3.6.1. Description

This routine adds the VFS node referenced by the second parameter to the VFS tree, the root of which is referenced by the first parameter.

1.3.6.2. Inputs

The two procedure inputs are:

- **root** – pointer to an existing root node of a VFS tree (repository); if this pointer is equal to NULL no operation is performed;
- **node** – pointer referencing an existing VFS node which can be of an `embedded_file`, an `embedded_script` or an `embedded_directory` kind; if the pointer does not point to a node of an appropriate kind, no operation is performed.

1.3.6.3. Outputs

None.

1.3.6.4. Fault messages

None.

1.3.7. Function *tdata_base_struct FindNode(tdata_base_struct root, tdata_base_struct path, pchar user, pchar password)*

1.3.7.1. Description

The role of the routine consists in finding the node, referenced by the object of the path kind (second parameter of the routine), in a VFS tree referenced by its root (first parameter of the routine). The process follows naturally the path object structure which is composed of a list of names of intermediary nodes of the **`embedded_directory`** kind, located on the path between the tree root and the searched node. The routine verifies the access rights at every stage of the search process (i.e. while visiting the intermediary directories). The credentials valid for the search process are provided by the third and fourth parameters of the routine.

1.3.7.2. Inputs

The four routine inputs are:

- **root** – pointer to an existing root node of a VFS tree (repository); if this pointer is equal to NULL no search is initialised;
- **path** – reference of an object which represents the path to be followed during the search process; the path is composed of a series of steps which reference by name the directory nodes lying between the tree root and searched object;
- **user** – pointer to the string which contains the user name; this parameter represent the credentials to be verified at every level of the searched tree;
- **password** – pointer to the string which contains the user password; this parameter completes the user name in credential verification.

1.3.7.3. Outputs

The routine always returns a reference to an object of **`tdata_base_struct`** type or a NULL pointer. If the search process succeeds the object referenced by the pointer is either the tree root, an embedded file, an embedded directory or an embedded script.

1.3.7.4. Fault messages

The routine fails when it does not return a reference to an existing tree node. No explicit textual message is generated. The reason for the failure can be two–fold:

- the node referenced by the path object does not belong to the tree; in this case the routine returns the NULL pointer;
- the search process was stopped at a stage because the presented credentials were not compatible with the authorisation required at that stage; in this case the routine returns the pointer to an object of **`the access_list`** kind.

1.3.8. Function *tdata_base_struct SearchNodeInWidth(tdata_base_struct root, pchar name)*

1.3.8.1. Description

This function looks for the node of a given name (string pointed by the second function parameter), in the VFS tree referenced by the pointer to its root (first function parameter). The result of the search is either a node reference or a NULL pointer. The search process is based on a width–first algorithm.

1.3.8.2. Inputs

The routine has two parameters:

- **root** – pointer to an existing root node of a VFS tree (repository); if this pointer is equal to NULL no search is initialised;
- **name** – pointer to a character string which identifies the node within the VFS tree.

1.3.8.3. Outputs

The routine returns NULL pointer if the node corresponding to the name was not found. In case of success the node reference is returned.

1.3.8.4. Fault messages

NULL pointer signifies the search failure or NULL tree reference.

1.3.9. Function *tdata_base_struct GetNodeRef(tdata_base_struct root, tdata_base_struct path)*

1.3.9.1. Description

This function checks whether a path provided as the second function parameter corresponds to a node in a tree referenced by the first function parameter.

1.3.9.2. Inputs

The routine has two parameters:

- root – pointer to an existing root node of a VFS tree (repository); if this pointer is equal to NULL no search is initialised;
- path – reference of an object which represents the path to be followed during the search process.

1.3.9.3. Outputs

The routine returns either the reference to an existing node or a NULL pointer.

1.3.9.4. Fault messages

Routine failures are represented by the NULL pointer returned. The NULL result is returned in two cases: either one of the input parameters is equal to NULL or the referenced node was not found. No explicit textual message is generated.

1.3.10. Procedure *void ProcessNode(tdata_base_struct node, tnameproc proc)*

1.3.10.1. Description

This routine provides the means of applying programmed processing on the node referenced by its first parameter. The operation to be applied to the node is programmed in a routine which is referenced by the second parameter.

1.3.10.2. Inputs

The routine has two parameters:

- **node** – pointer to an existing node of a VFS tree (tree root, embedded file, embedded script or embedded directory); if this pointer is equal to NULL no operation is performed;
- **proc** – reference of a routine which allows the user to program the operation to be performed on the node; the data type of this parameter is defined as follows:

  typedef int(* tnameproc)(pchar,int,int)

No processing is done if this parameter is NULL.

1.3.10.3. Outputs

None.

1.3.10.4. Fault messages

None.

1.3.11. Function *pchar InstallFileContent(tdata_base_struct root, tdata_base_struct path, pchar contents, unsigned short length)*

1.3.11.1. Description

This function links a memory area to a VFS tree file; the file location in the tree is referenced by the path object passed by one of the parameters of the routine.

1.3.11.2. Inputs

The routine has four parameters:

- **root** – pointer to a root of an existing VFS; if this pointer is equal to NULL, the operation has no effect and the function returns NULL;
- **path** – path object which localises the node within the VFS tree; if this pointer is equal to NULL, the operation has no effect and the function returns NULL;
- **contents** – pointer to a contiguous memory area storing the bytes which form the body of the embedded object (page, image, Java byte code);
- **length** – parameter which stores the length of the region pointed by contents; this parameter is zero for null terminated character strings.

1.3.11.3. Outputs

The routine returns the pointer to the region if the installation process was done correctly. In any other case the routine returns NULL.

1.3.11.4. Fault messages

If the input parameters are incongruent or if the referenced node cannot be found, the routine returns NULL. No explicit textual message is generated.

1.3.12. Procedure *void InstallAuthorisation(tdata_base_struct realm, pchar username, pchar password)*

1.3.12.1. Description

This procedure provides the means of configuring access control to a VFS tree or to an embedded directory (VFS sub–tree). The restricted VFS region is referenced by the first parameter of the routine while the second and third parameters represent the security attributes stored in a record verified by the Find Node procedure.

1.3.12.2. Inputs

The function has three parameters:

- **realm** – pointer to a root of an existing VFS or to an existing tree node of the *embedded_directory* kind; if this pointer is equal to NULL, the operation has no effect and the function returns NULL;
- **username**– pointer to the string which contains the user name; this parameter represent the credentials to be verified at every level of the searched tree;
- **password** – pointer to the string which contains the user password; this parameter completes the user name in credential verification.

1.3.12.3. Outputs

None.

1.3.12.4. Fault messages

None.

1.3.13. Function *tdata_base_struct AppendStep(tdata_base_struct path:, pchar name:)*

1.3.13.1. Description

This function is used for the successive construction of path objects. The object stores the access track which leads from a tree root to a node within this tree. The function implements add the step named by the second parameter to the path referenced by the first parameter.

1.3.13.2. Inputs

The function has two inputs:

- **path** – path object which localises the node within the VFS tree; if this pointer is equal to NULL, the operation has no effect and the function returns NULL;
- **name**– pointer to the string which contains the name of the step to be appended to the path.

1.3.13.3. Outputs

The function returns the reference of the newly created object having the new step appended to the one provided by the first parameter.

1.3.13.4. Fault messages

If the function fails to append the new step, NULL pointer is returned. No explicit textual message is generated.

1.3.14. Function *pchar RecoverResName(tdata_base_struct path)*

1.3.14.1. Description

This function, used in the process of constructing request environment variables, recovers the last step of the path object passed by the input parameter and considers it as the resource name, i.e. the name of a VFS node to which the HTTP request in which the path was included applies.

1.3.14.2. Inputs

This function has a single input which contains the reference to a path kind object.

1.3.14.3. Outputs

The function returns the pointer to the storage area in which the name of the last step of the path is stored.

1.3.14.4. Fault messages

When the NULL pointer is provided as the parameter the function returns NULL. No explicit textual message is generated.

1.3.15. Function *pchar PathToString(tdata_base_struct path)*

1.3.15.1. Description

This function is used mostly for debugging purposes. It converts the path object referenced by the first parameter to a printable character string which has the following syntax:

“step1_name/step2_name/step_name*……*../stepn_name”

1.3.15.2. Inputs

The unique input of the function contains the reference of the path object.

1.3.15.3. Outputs

The function returns the pointer to the character string, allocated in the system heap, which contains the printable representation of the path.

1.3.15.4. Fault messages

The NULL pointer is returned if the reference of the path object is NULL. No explicit textual message is generated.

1.3.16. Function *tdata_base_struct StringToPath(pchar pathstring)*

1.3.16.1. Description

This function provides the operation which is the reverse of the function *PathToString*. It converts the printable representation of the path to the path object.

1.3.16.2. Inputs

The unique parameter of the function is a pointer to the character string. The contents of this string should strictly observe the following syntax:

"step1_name/step2_name/step_name*……*../stepn_name"

1.3.16.3. Outputs

The function returns the reference of an object of the path kind. The path object is created dynamically on the system heap.

1.3.16.4. Fault messages

The NULL pointer is returned when the input string is NULL or when it does not observe the imposed syntax. No explicit textual message is generated.

1.3.17. Procedure *void PrintPath(tdata_base_struct path):*

1.3.17.1. Description

This routine serves to debug server applications. Its role consists in printing on the standard output the character string which represents the path object provided as input. The printout has the following syntax:

"step1_name/step2_name/step_name*……*../stepn_name"

1.3.17.2. Inputs

The unique input of the function is the pointer to the path object.

1.3.17.3. Outputs

None.

1.3.17.4. Fault messages

If the input does not reference a valid object of the path kind, the routine does not print anything.

1.3.18. Procedure *void PrintNode(tdata_base_struct node)*

1.3.18.1. Description

This routine serves to debug on server applications. Its role consists in printing the data describing the input object on the standard output. The format of printed information is as follows:

- if the input is the NULL pointer, the printed message is:

  `NULL FILE POINTER`;
- if the input is an embedded file or embedded script node, the printed message is:

`FILE` *file name*	if the file name is provided (not NULL pointer) or
`ANONYMOUS FILE`	if the file name is not provided (NULL pointer);
- if the input is an embedded directory node, the printed message is:

`DIRECTORY` *directory name*	if the directory name is provided (not NULL pointer) or
`ANONYMOUS DIRECTORY`	if the directory name is not provided (NULL pointer);
- if the input is the tree root node, the printed message is:

  `ROOT`;
- in any other case no message is printed on the output.

1.3.18.2. Inputs

Reference to the node to be printed out.

1.3.18.3. Outputs

None.

1.3.18.4. Fault messages

None.

1.3.19. Procedure *void PrintRepository(tdata_base_struct root)*

1.3.19.1. Description

This routine is used for debugging the VFS generated on the server machine. Its role consists in visiting a VFS tree from the root till leaves and printing out the messages concerning the visited nodes. The tree is visited in the depth–first order. The messages are printed on the standard output on the server's platform. This routine calls the PrintNode routine from the same module so it inherits the messages from the latter. These messages are wrapped into supplementary printouts which are:

- Each repository printout starts with the message:

  `REPOSITORY` *repository name* **`STARTS`** if the repository name is known or

  `REPOSITORY unnamed STARTS` if the repository name is not known. This message is followed by the series of printouts for visited nodes.

- For the repository root and for each embedded directory node the message generated by the PrintNode routine is completed by the line representing the contents of the access list:

`ACCESS LIST IS EMPTY`	if the access list is empty (pointer set to NULL) or
`ACCESS LIST`	followed by the series of pairs of identifiers representing the authorised user and its password if the access list is not empty.

- The repository printout ends with the message:

`REPOSITORY` *repository name* **`ENDS`**	if the repository name is known or
`REPOSITORY unnamed ENDS`	if the repository name is not known.

1.3.19.2. Inputs

Pointer to the root of a VFS tree.

1.3.19.3. Outputs

None.

1.3.19.4. Fault messages

If the input node is NULL, no message is printed on the output.

2. PACKAGE SERVER ENGINE

The package regroups the modules which implement the operation of the server on the global level such as management of server's main loop, adaptation to the network, management of response generation and management of request's persistence. It is composed of four modules:

- module **`servengine`** – regroups the routines implementing the main server loop;
- module **`sendback`** – is in charge of servicing the objects;
- module **`sockinterf`** – adapts the server to the network interface;
- module **`data_base_module`** – implements the persistence of data sent from the client via the request PDU.

The mutual relationships are represented by the diagram below.

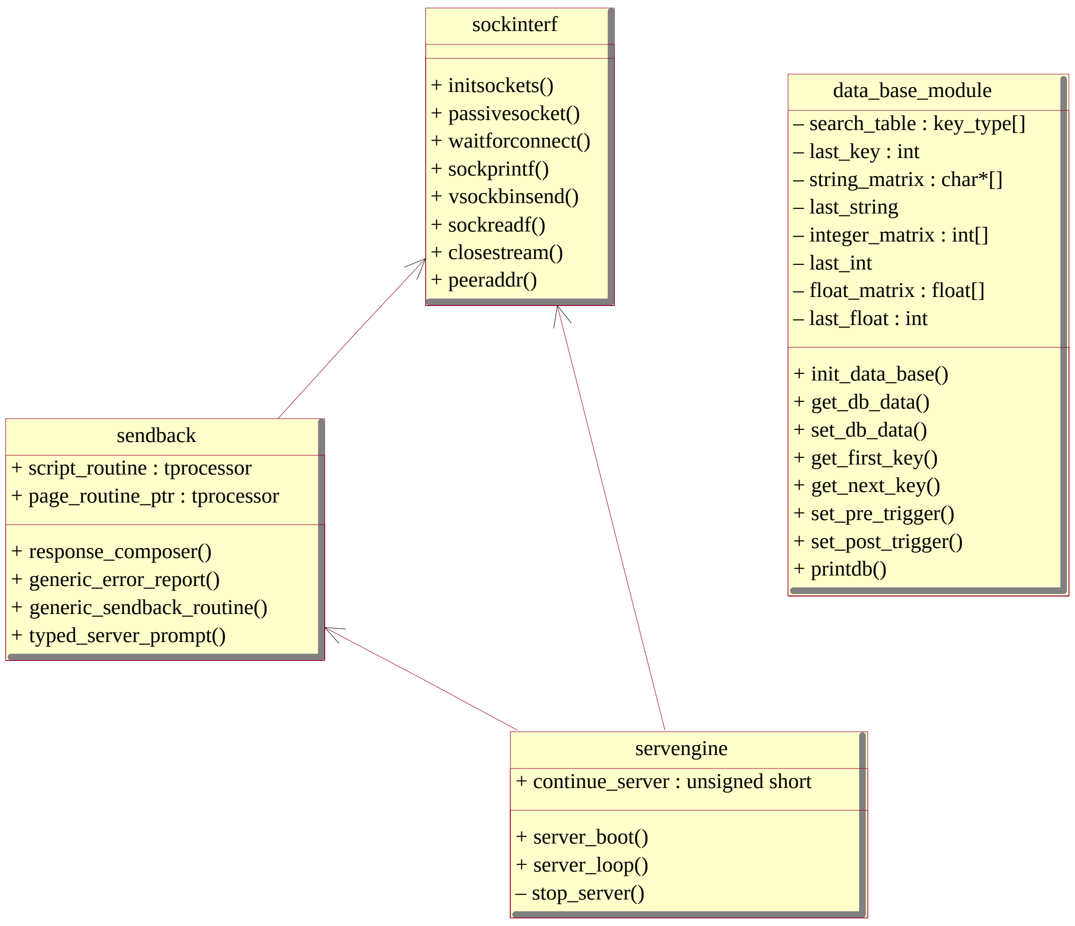


Figure 3.2 – Module relationships within the package *Servengine*

One can easily observe that only three out of four modules co–operate: **`sendback`**, **`servengine`** and **`sockinterf`**. The fourth module which is in charge of implementing the request data persistence, is stand–alone and does not provide directly any services to the three others. It is put in this package for classification purposes.

2.1. Module Servengine

The module **`servengine`** implements the driving mechanism of the server. It provides two routines which both execute the algorithm of the iterative server communicating with the client via a TCP/IP socket. This algorithm is composed of a four–stage execution loop preceded by an initialisation step (for control flow see Figure 3.3).

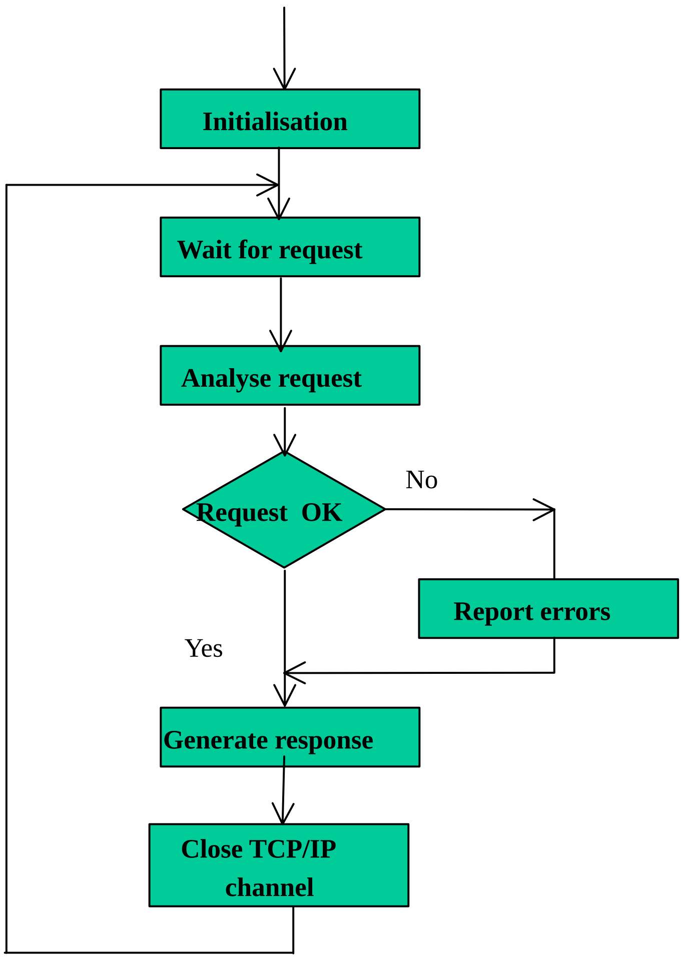


Figure 3.3 – Flow diagram of the basic server loop

The operation of the loop is driven by arrivals of request PDUs on the TCP/IP socket. PDUs are recovered from the socket and passed to a routine which analyses their contents. The role of such a routine consists in verifying the structure of the PDU and in extracting the information necessary for further server loop operations.

This routine decides if the PDU is well–formed or if there is an error in its structure. In case of error detection, the second processing stage, reporting the error, is activated. In the case of correct reception the third stage is started. This stage consists in generation of the server’s response corresponding to the request (correct or erroneous).

The single round of the loop is terminated by a clean–up routine.

The present software provides implementations for each processing stage of the loop described previously. Their descriptions can be found in § 2.2. and 3.1. The application programmer using this software is free to develop his own procedures implementing the basic loop stages. The only constraint imposed on this development is the signature of such routines as described in the following sections.

Both routines provided in the package implement this basic loop with more or less flexibility left to the application programmer, or in other terms, the application programmer is required to do more or less programming.

2.1.1. Procedure *void server_loop(unsigned short port_nr, tcallback analyse_routine_ref, tcallback response_routine_ref, tcallback error_report_ref, tcallback cleanup_routine_ref)*

2.1.1.1. Description

This routine which implements the basic server loop described in the introductory section, provides the application programmer with the possibility of choosing the implementation details. By an appropriate selection of parameter values the user can easily modify the behaviour of the server operation and can build his own implementation for each processing stage of the server.

2.1.1.2. Inputs

The routine has five parameters. The first is of the **`unsigned short`** type while the others are of **`the tcallback`** type, which is exported by the module. The definition of the type is as follows:

typedef unsigned short tcallback (int)

The semantics of the input parameters are the following:

- **unsigned short port_nr** – this input provides the value of the TCP port on which the server will operate;
- **tcallback analyse_routine_ref** – this input parameter provides the reference of a routine implementing the incoming PDU analysis;
- **tcallback response_routine_ref** – this input parameter provides the reference of a routine implementing the generation of server’s response to the request;
- **tcallback error_report_ref** – this input parameter provides the reference of a routine implementing the server reaction in case of error detected during the analysis phase;
- **tcallback cleanup_routine_ref** – this input parameter provides the reference of a routine implementing the termination phase of the basic loop.

Values passed by the four last input parameters can correspond either to the user programmed routines or to one of the specialised routines described in this manual. It is important to state that the user–provided routines have to have the signatures compatible with the type **`tcallback.`**

2.1.1.3. Outputs

The routine does not provide any output values.

2.1.1.4. Operational processing

The control flow of the routine follows the diagram in Figure 3.3. In the initialisation phase the routine implements the following operations:

- attempts to create a TCP type socket and assigns it the TCP port number passed by the first parameter of the routine; if socket creation fails the routine exits and the server loop is not started;

- verifies the values of input parameters corresponding to the references of the routine implementing the four phases of server operation (parameters 2, 3, 4 and 5 of the type `tcallback`); if a parameter value is set to NULL the corresponding phase of the server loop is implemented by the predefined routine according to the following table.

Server loop phase	Corresponding Predefined Routine	Implemented in module
Analysis	ParseFromSocket	http_process (§ 3.1.)
Error reporting	generic_error_report	senback (§ 2.2.)
Response generation	generic_sendback_routine	sendback (§ 2.2.)
Loop closing	EndParsing	http_process (§ 3.1.)

Table 3.3 – Server loop phase, corresponding routine and module

The loop is started after the successful termination of the initialisation phase and, in principle, never ends without external intervention. This intervention is possible since the loop is controlled by an exported variable.

unsigned short continue_server.

The application programmer can implement the server stopping routine by setting this variable to 0.

2.1.1.5. Fault messages

None.

2.1.2. Procedure *void server_boot(unsigned short port_nr)*

2.1.2.1. Description

This routine implements the basic server loop without providing the possibility to choose implementation of the server loop phases. By using this routine the application programmer can only choose the port on which the server will operate. All server phases are implemented by the predefined routines listed in Table 3.3.

2.1.2.2. Inputs

The single input parameter of the routine provides the port number on which the server will operate.

2.1.2.3. Outputs

The routine does not provide any output values.

2.1.2.4. Operational processing

This routine calls the routine **`server_loop`** with the first parameter set to its input parameter and four others set to NULL. This mode of calling **`server_loop`** will force the operation of the server with the predefined implementation of the processing phases, and frees the user from the burden of programming. The procedure is a recommended entry point for software users who wish to concentrate their efforts on application development.

2.1.2.5. Fault messages

None.

2.2. Module *Sendback*

The module `sendback` contains the routines provided for the standard implementation of the response composition phase of server operation. It exports:

- two variables:
  - send_script_routine,
  - send_page_routine.
- four routines:
  - response_composer,
  - generic_sendback_routine,
  - generic_error_report,
  - typed_server_prompt.

The module also exports one type of definition:

typedef int tprocessor(int,…)

The type is used to qualify `response_composer` input parameters.

These six items provide an interface which allows the user to organise the mechanisms of object handling by the embedded server.

2.2.1. Variables exported by the module

The two variables: `send_script_routine` and `send_page_routine`, are defined as follows:

int(*send_script_routine)(int,tdata_base_struct)

int(*send_page_routine) (int,tdata_base_struct)

They are references of the routines which will be used by the server loop to implement the two mains operations:

- response to requests of cgi script execution – `send_script_routine;`
- response to requests for sending internal server objects: pages, images and applets – `send_script_routine;`

The user is expected to provide, via these variables, the addresses of the routines specially designed for its application. By default the variables are set to NULL. They are expected to receive the appropriate values during the initialisation phase of the server loop. If the initialisation is not done properly, the server will systematically answer any client request by sending a standard error response described in the following section.

2.2.2. Procedure *int response_composer(int socket_id, tprocessor scrproc, tprocessor pproc, tprocessor dproc)*

2.2.2.1. Description

This routine implements the basic elements of the process ruling the composition of responses to clients' requests received by the server. It provides the application programmer with the possibility of choosing some implementation details while it sets others. By an appropriate selection of parameter values, the user can easily modify the routine behaviour and can provide his own implementation of service response operations.

2.2.2.2. Inputs

The routine has three parameters. The first is of the **`int`** type and the two others are of the **`tprocessor`** type defined in § 2.2.

The semantics of the input parameters are the following:

- **int socket_id** – this input provides the identifier of the active socket through which the remote client is connected to the server;
- **tprocessor scrproc** – this input parameter provides the reference of a routine implementing the execution process of CGI scripts;
- **tprocessor pproc** – this input parameter provides the reference of a routine implementing the generation process of passive objects (pages, images and applets);
- **tprocessor dproc** – this input parameter provides the reference of a routine implementing server response when the requested object is a domain (file directory).

Values passed by the second, the third and the fourth input parameters can correspond either to the user programmed routines or to one of the specialised routines described in this manual. It is important to state that the user–provided routines must have the signatures compatible with the type **`tprocessor.`**

2.2.2.3. Outputs

The output provided by the routine reflects the result of the response generation process. In general a positive, non–zero result corresponds to a situation in which a server object was correctly sent to the client. Such an object can correspond to a requested page, image or applet, to an activated script or to an error signalling page. The negative or zero result means that the response generation was not successful and that nothing was sent back to the client.

2.2.2.4. Operational processing

The operation of the routine follows the flow diagram shown in Figure 3.4.

As it can be seen, the processing is organised in three steps:

1. identification of object processors

   First step of the routine consists in choosing between two exclusive processing threads: one for CGI script activation and one for passive object transmission. The processors are to be used in subsequent steps of the routine. The choice is based on the values passed via the second and the third input parameter.

If the value of the second parameter is set to NULL, CGI script execution will be realised by the routine whose address is passed via the external variable **`send_script_routine`**. If this value corresponds to the entry point of a user–provided routine, it will be used to rule script execution. If the value of the third parameter is set to NULL, the passive objects service (pages, images, applets) will be performed by the routine whose address is passed via the external variable **`send_page_routine`**. If this value corresponds to the entry point of a user provided routine, it will be used to control the response generation.

2. identification of type of object to service

   The second step of the routine consists in identifying the type of object, i.e. verifying whether the requested action consists in activating a script, sending a passive object or requesting a directory listing. The outcome of this choice activates an appropriate processor routine.

3. object search and identification of action to be executed.

   In the third phase the requested object is searched for. If it is not found, an error page is selected and prepared for delivery to the client. If the object is found and is not submitted to access authentification, the previously chosen processor is activated. If the object is submitted to access authentification, the challenge procedure is started by selection of an appropriate response PDU.

At the end, the selected action is executed and its result is returned as the result of the routine.

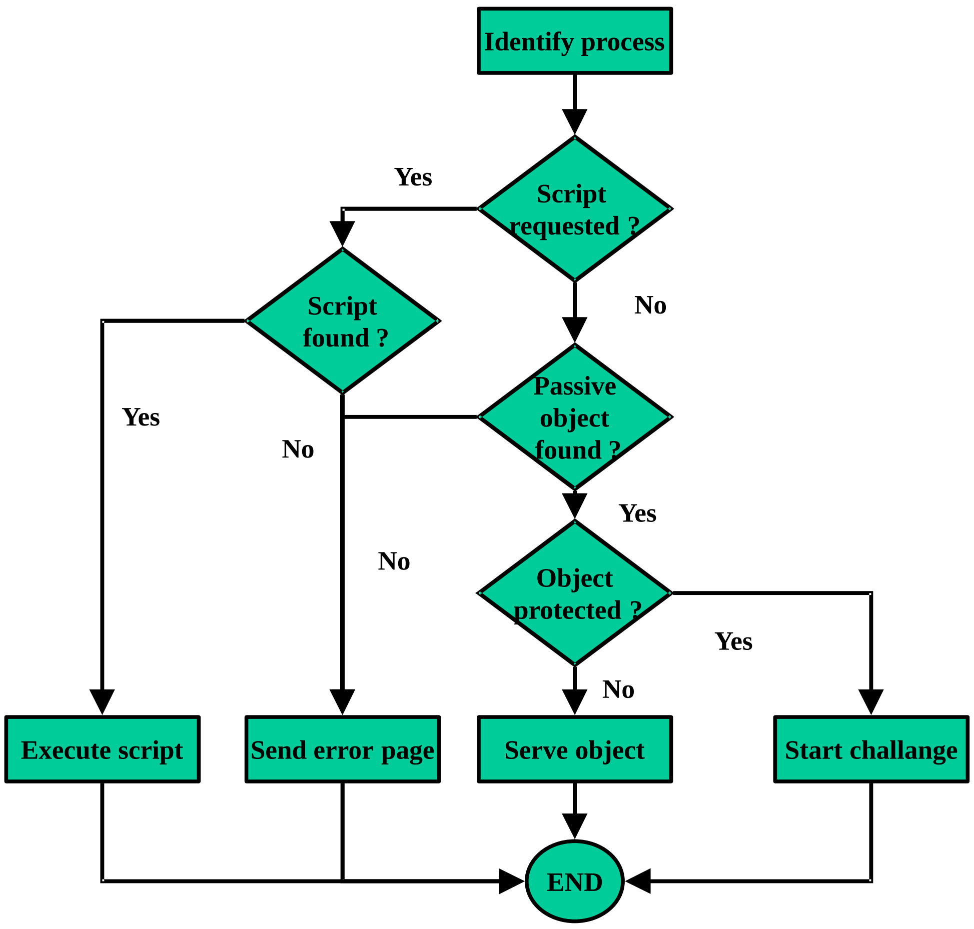


Figure 3.4 – Flow diagram of the response–composing routine

2.2.2.5. Fault messages

In the case of failure of the last step of the routine’s algorithm, an error message can be sent as a special value of the return code. This message is produced by the routine which writes the response PDU to the network and is inherited from the module implementing socket interface to TCP/IP. In the majority of cases a negative or null value corresponds to a transmission failure.

2.2.3. Procedure *unsigned short generic_sendback_routine(int socket_id)*

2.2.3.1. Description

This routine serves as a wrapper to the previously described **`response_composer`** routine, which is invoked with the second and third parameters set to NULL. By choosing these parameter values, the routine forces the operation mode for the invoked **`response_composer`** processor, in which the choice of the object servicing routine is mode via two pointers **`send_script_routine`** and **`send_page_routine`** (see § 2.2.1.). This routine is proposed à priori as an implementation choice for the response composition phase in the server’s basic loop (see § 2.1.2.).

2.2.3.2. Inputs

The only input parameter of the routine corresponds to the identifier of the socket on which the connection with the client is established. This parameter is passed over to the wrapped **`response_composer`** routine.

2.2.3.3. Outputs

The routine returns the value which is returned from the wrapped **`response_composer`** routine. For the meaning of the returned values refer to § 2.2.2.3.

2.2.3.4. Operational processing

Routine processing is limited to the invocation of the **`response_composer`** routine with the second and third parameters set to NULL.

2.2.3.5. Fault messages

The routine returns the same condition codes as described in the **`response_composer`** routine (see § 2.2.2.5.).

2.2.4. Procedure *unsigned short generic_error_report(int socket_id)*

2.2.4.1. Description

This routine is provided in order to free the user from the programming burden required for implementation of the error reporting phase in the basic server loop introduction (see § 2.1.).

2.2.4.2. Inputs

The only input to the routine provides the socket identifier through which the server is communicating with the remote client.

2.2.4.3. Outputs

The output generated by the routine reflects the result of the sending error page operation to the client. If the operation is successful the returned value is positive and non–zero. Negative values correspond to transmission errors detected by the socket interface layer.

2.2.4.4. Operational processing

This procedure prepares a standard error page which informs the client on an internal server malfunction without any particular problem description.

2.2.4.5. Fault messages

The negative values returned by the routine show the malfunction of the network and mean that the error page has not attained the client.

2.2.5. Procedure *int typed_server_prompt(int socket_id, char* res_type)*

2.2.5.1. Description

This routine is used to generate the standard server header string used in the composition of successful response PDUs returned by the server to its remote clients. It is strongly recommended to use this method of server identification in any user provided routines.

2.2.5.2. Inputs

The first input parameter provides the identifier of the socket via which the server is communicating with the remote client. The second input is a character string which identifies the type of resource requested by the client.

2.2.5.3. Outputs

The output generated by the routine reflects the result of the operation of sending the header string to the client. If the operation is successful the returned value is positive and non–zero. Negative values correspond to transmission errors detected by the socket interface layer.

2.2.5.4. Operational processing

At first, the routine tests whether the value of the second input parameter is equal to NULL. If this is the case, the execution terminates and the result –1 is returned to the caller. If the value is different from NULL the execution continues. The next step consists in analysing the contents of the string passed via the second input parameter. If the string pointed to by the second parameter is “gif” of “jpeg”, the generated server header is as follows:

```
HTTP/1.0 200 0K <cr><lf>
Server: FIPWEB /0.9 <cr><lf>
Content–type: image/gif <cr><lf>
<cr><lf>
```

If the string pointed to by the second parameter is different from the values indicated above, the server header is as follows:

```
HTTP/1.0 200 0K <cr><lf>
Server: FIPWEB /0.9 <cr><lf>
Content–type: text/html <cr><lf>
<cr><lf>
```

2.2.5.5. Fault messages

The negative values returned by the routine show the malfunction of the network and mean that the header string has not attained the network. The output value set to –1 may also mean that the NULL string was routed via the second input parameter.

2.3. Module Sockinterf

The purpose of this module is to provide the routines which directly use network–oriented services with an interface independent from the type of the stack used on the implementation platform. The interface exported by this module wraps the basic calls to a socket–type network interface. Providing this interface porting the whole software from one communication platform to another, requires only re–implementation of this module. The module provides the following routines:

● initsockets	– initialises socket layer,
● passivesocket	– creates the passive (server type) socket,
● waitforconnection	– waits for incoming connection requests on a secondary socket,
● closestream	– closes the passive socket,
● sockprintf	– formatted write to socket,
● vsockbinsend	– binary write to socket,
● sockreadf	– read from socket,
● peeraddrstr	– identification of the address of the remote peer.

In its current version the module adapts the rest of the software to three networking packages: the SockApi package of Alstom, the BSD Sockets implementation of SUN Solaris and the Winsock2 socket implementation of Microsoft.

2.3.1. Procedure *int initsockets(void)*

2.3.1.1. Description

Some implementations of socket API, like Microsoft's Winsock2, require an explicit initialisation procedure. This routine provides the wrapper of the initialisation operation for such implementations. For the portability purposes, it is recommended to call this routine any time when network operation should be restarted.

2.3.1.2. Inputs

None.

2.3.1.3. Outputs

The routine returns an integer type result which can be only one of two values 1 or –1. A successful initialisation operation corresponds to the value 1.

2.3.1.4. Fault messages

A returned value equal to –1 signifies the routine's failure.

2.3.2. Procedure *int passivesocket(unsigned short port_nr,char* type,int quelen)*

2.3.2.1. Description

This routine provides the wrapper for the series of operations which lead to the creation of a server socket bound to an IP address of the platform, completed by the port number provided via input parameters. When the routine returns successfully the socket is created and ready to accept the external connections.

2.3.2.2. Inputs

The routine has three inputs which are:

- **port_nr** – unsigned char type integer signifying the TCP/UDP port number to which the socket is linked;
- **type** – pointer to character chain holding the protocol designation, it should be either “`tcp`” or “`udp`”;
- **quelen** – integer number signifying the queue length reserved for buffering the incoming connection requests.

2.3.2.3. Outputs

The value returned by the routine is either the identifier of the socket bound to the chosen address and ready to accept the connections or an error condition code which identifies the eason for incorrect network operation. The socket identifier returned from the routine is always positive. The negative result always means a malfunction.

2.3.2.4. Operational processing

The routine calls the operations on sockets in the following order:

1. Socket creation of a given type; the socket type can be either SOCK_DGRM or SOCK_STREAM; the first type is created when the second routine’s parameter contains the string “udp”. In any other case the socket created is of the SOCK_STREAM, i.e. TCP type. If socket creation fails, the routine exits with an error code which is a non positive value. An error message accompanies this failure detection.
2. Once the creation of the socket succeeds, one attempts to bind it to a network address; this operation makes a call to the **`bind`** primitive; the part of the address relative to the IP of the network is taken from the platform while the TCP/UDP port number is passed via the routine’s parameters. If the conditions of address binding are accepted, the routine continues. In the opposite case an error message is issued and the routine exits.
3. If the socket is of the SOCK_STREAM type and if binding of the socket to an address succeeds, one attempts to make it listen to incoming connection requests, and this call to the **`listen`** operation makes use of the third parameter of the routine which establishes the length of the queue used for buffering the incoming requests for connection. If this call succeeds, the routine exits leaving the created socket passively waiting for the request for connections. In case of failure, an error message is produced before the routine’s exit.

2.3.2.5. Fault messages

In case of negative reaction from the network API, this routine prints messages on the error output. The messages follow a certain format but their definite shape depends on the implementation platform. There are three types of messages:

1. socket creation failure:

   `can’t create socket` –<platform dependent string>

2. bind to address failure:

   `cant bind the socket to port nr.` <port nr> –<platform dependent string>

3. listen activation failure:

   `can’t listen to socket on port nr` <port nr> –<platform dependent string>

2.3.3. Procedure *int waitforconnect(int main_sock)*

2.3.3.1. Description

This routine provides the platform–independent interface to the **`accept`** operation on the primary socket followed by retrieval of the remote peer address.

2.3.3.2. Inputs

The only input parameter is the identifier of the primary server socket which waits for the requests for connections.

2.3.3.3. Outputs

The routine returns either the identifier of a secondary socket created as the result of connection request acceptance, or an error code.

2.3.3.4. Operational processing

The control flow of the routine makes use of a blocking call to the **`accept`** primitive to stop the main server loop while waiting for the connection request from the remote client. Connection acceptance is followed by the creation of a secondary socket. The identifier of this socket, returned by the call to **`accept,`** is then held to be returned further on as the routine result. Failure of the call to **`accept`**, which gives the negative or zero result, makes the routine return immediately. The success of **`the accept`** call provides the retrieval of the remote peer address to a local data structure. This address can be retrieved by the call to the ***peername*** routine.

2.3.3.5. Fault messages

None.

2.3.4. Procedure *int closestream(int sock_id)*

2.3.4.1. Description

This routine is a wrapper of the operation when closing the socket.

2.3.4.2. Inputs

The only input of the routine is the identifier of the socket to be closed.

2.3.4.3. Outputs

The routine transfers the result of the call of the **`closesocket`** operation.

2.3.4.4. Fault messages

None.

2.3.5. Procedure *int sockprintf(int sock_id,char*xtempl,…)*

2.3.5.1. Description

This routine is used to send a block of characters to the network via a secondary socket. The transfer mode resembles a formatted write of a list of items to a file.

2.3.5.2. Inputs

The number of inputs to this routine can vary. There are at least two inputs:

- sock_id – socket identifier;
- xtempl – character string which is constructed according to rules set for the format specification in the `printf, fprints, sprintf,` etc. routines.

The routine can optionally have a set of inputs corresponding semantically to the items constructing the format string.

2.3.5.3. Outputs

The output from the procedure provides the result of the transmission attempt. It gives either a number of characters written to the socket or an error code resulting from the operation.

2.3.5.4. Operational processing

The operation of the procedure progresses in many steps. In the first step the format (second input) and the list of ***free parameters*** (all the starting parameters remaining from the third input) are used to generate a string which is stored in an internal buffer. It is important to know that the result of this operation is a string having less than a fixed number of characters (in this case – less then 32768 characters). After this operation the string is sent via socket and the transmission result is returned to the caller.

2.3.5.5. Fault messages

Failure of the routine's operation is reflected by the value returned by it. In case of successful transmission the result should be positive and should correspond to the number of characters sent. Negative or zero values correspond to transmission errors.

2.3.6. Procedure *int vsockbinsend(int sock_id,char* string,unsigned short length)*

2.3.6.1. Description

This routine is used to send a block of bytes of a given length to the network via a secondary socket. The transfer mode corresponds to a non formatted write of a character stream to a file.

2.3.6.2. Inputs

The routine has three inputs:

- **sock_int** – socket identifier;
- **string** – character string written to the socket;
- **length** – length of the string passed by the second parameter.

2.3.6.3. Outputs

The output provides the result of the transmission attempt. It gives either a number of characters written to the socket or an error code resulting from the operation.

2.3.6.4. Fault messages

The operation failure is reflected by the value returned by it. In the case of successful transmission the result should be positive and should correspond to the number of bytes sent. Negative or zero values correspond to transmission errors.

2.3.7. Procedure *int sockreadf(int sock_id,char* buffer, int size)*

2.3.7.1. Description

This routine is a wrapper of the synchronous `recv` operation.

2.3.7.2. Inputs

The routine has three inputs:

- sock_id: socket identifier;
- buffer: pointer to the buffer which receives the characters from the socket;
- size: length of the buffer.

2.3.7.3. Outputs

The output provides the result of the reception from the socket. It gives either a number of characters received via the socket or an error code resulting from the failure of the operation.

2.3.7.4. Fault messages

The operation failure is reported by the value returned by it. In the case of successful reception the result is positive and corresponds to the number of characters sent. Negative or zero values correspond to the reception anomalies.

2.3.8. Procedure *void peeraddrstr(char* buffer)*

2.3.8.1. Description

This routine recovers the distant peer address retreived previously called the **`waitforconnect`** routine.

2.3.8.2. Inputs

Reference of an existing character buffer ready to receive the remote user address in the form of a character string.

2.3.8.3. Outputs

None.

2.3.8.4. Fault messages

None.

2.4. Module data_base_module

This module handles the persistence of data arriving via the request PDUs. The data arriving in the server in HTML form are structured as a series of name/value pairs. These pairs are extracted from the request PDU by the parsing procedure and are placed within the table managed by the current module. The extracted data can have one of three formats: integer numbers, floating point numbers or character strings. The module provides the following routines:

- init_data_base – initialises tables which hold the data,
- get_db_data – retrieves a value from data tables,
- set_db_data – sets a value in data tables,
- get_first_key – returns the name of the first variable stored in the tables,
- get_next_key – returns the name of the next variable following the one passed by the parameter.

These routines provide access to data stored in the table structure which is opaque to the user. The user retrieves and sets the values of the data record by identifying it via its *key* which is a printable character string. The identification of a data record via its key is unambiguous since the internal mechanisms of the module guarantee the uniqueness of the data key value within the tables.

In order to program the retrieval and storage of data in the internal tables the module exports the data type **`tdb_result`** defined as follows:

```
        typedef struct tagdb_result {
                                int result;
                                union{
                                     char* string;
                                     int integer;
                                     float   real   }   value;   }
        tdb_result;
```

This type is used especially by **`get_db_data`** and **`set_db_data`** routines.

2.4.1. Procedure *void init_data_base(void)*

2.4.1.1. Description

This routine resets the internal data store. It clears all internal tables and resets all data records active before this call.

2.4.1.2. Inputs

None.

2.4.1.3. Outputs

None.

2.4.1.4. Fault messages

None.

2.4.2. Procedure *void get_db_data(char* name, tdb_result* result)*

2.4.2.1. Description

The role of this routine consists in retrieving the data value corresponding to the identification key.

2.4.2.2. Inputs

The only input to the routine is the key identifying the data which is to be retrieved. The key, a string of printable characters, is furnished via the first parameter of the routine.

2.4.2.3. Outputs

The output is obtained via the second parameter. This parameter is a pointer which should reference an existing structure of the `tdb_resul` type.

The structure is composed of two fields, `result` and `value`. The first field indicates the kind of response obtained. It can take the following values:

- **unknown** – corresponding to the value –1,
- **string** – corresponding to the value 0 (symbolically `c_string`),
- **integer** – corresponding to the value 1 (symbolically `c_int`),
- **floating point** – corresponding to the value 2 (symbolically `c_float`).

The second field transfers the value retrieved from the tables. Its type depends on the value of the first field according to the following relationship:

result field value	variant name	value field type
unknown	not concerned	undetermined
c_string	string	char*
c_int	integer	int
c_float	real	float

Table 3.4 – Relationship between the first and second field values

The existence of the structure passed as the second parameter is a pre–condition for the routine's correct operation. If the user does not follow this condition, the routine fails and the failure mode is unpredictable.

2.4.2.4. Operational processing

The first phase of the routine's algorithm consists in searching a record identified by the key passed by the routine's first parameter. If the search process fails, the ***result*** field of the output parameter is set to –1 and the value of the second parameter is not determined.

If the data record corresponding to the key is found, its type is determined. If the type corresponds to one of three recognised types, the ***result*** and the ***value*** field of the output parameter are set to the values stored within the record. In the opposite case the ***result*** is set to –1 (**unknown**) and the ***value*** remains undetermined.

2.4.2.5. Fault messages

The routine fails when the second parameter is NULL. No explicit error message is sent and this type of failure can result in the system exception provoked by access to the illegal memory address.

2.4.3. Procedure *int set_db_data(char* name, unsigned char val_type, void* value)*

2.4.3.1. Description

The role of this routine consists in modifying the data value which corresponds to the identification key.

2.4.3.2. Inputs

The routine has three input parameters:

- **data key** – string of printable characters identifying the data record to be modified,
- **value type** – integer of `unsigned char` type determining the type of data modified,
- **data value** – pointer to anonymous type `void*` conveying the data to be set into the record.

2.4.3.3. Outputs

The routine returns the value which identifies the result of the data record modification. If the modification is successful the routine returns the type equal to its second parameter. In the opposite case the returned value is –1.

2.4.3.4. Operational processing

The first phase of the routine's algorithm consists in searching a record identified by the key passed by the routine's first parameter. If the search process succeeds, the value type of the found record is tested. If the type of the found record is compatible with the type specified by the second parameter of the routine, the new data value is set. The value transferred to the data record corresponds to the value referenced by the third input parameter of the routine. If the types are not compatible, the data is not set and the routine returns the –1 value. In the case of successful value transfer the routine returns the value corresponding to the type of the record. Type compatibility is determined according to the following table:

record type vs routine parameter	c_int	c_float	c_string	any other
c_int	yes	yes	no	no
c_float	yes	yes	no	no
c_string	no	no	yes	no
any other	no	no	no	no

Table 3.5 – Compatibility of the record type and the routine parameter

The value modification changes the type of the data record when compatibility is set to ”no”. It is important that the third routine parameter references an existing variable having the right type. If the user does not respect this condition, the routine fails and the failure mode is platform–dependent. If the key value does not specify any existing record, the routine attempts to create the new data record and sets its type and value according to input parameters. The returned result corresponds to the type of newly created record.

If the attempt to create the new record fails, the routine returns the –1 value.

2.4.3.5. Fault messages

The routine fails when the third parameter is NULL. No explicit error message is sent and this type of failure ends up with the system exception provoked by the access to an illegal memory address.

2.4.4. Procedure *char* get_first_key(void)*

2.4.4.1. Description

The purpose of this routine is the identification of the data record key values in the internal table.

2.4.4.2. Inputs

None.

2.4.4.3. Outputs

The routine returns a reference of a character string. If the data table is not empty, this character string corresponds to the key value of the first data record. In the case of an empty table, a NULL pointer is returned.

2.4.4.4. Fault messages

None.

2.4.5. Procedure *char* get_next_key(char* ref_name)*

2.4.5.1. Description

This procedure, while called repetitively, allows the user to obtain the list of keys of all records stored in the data table. The single call provides the user with the value of the key immediately following the record whose key is referenced as the routine parameter.

2.4.5.2. Inputs

The only input to the routine is the character string corresponding to a key of a record within the data table.

2.4.5.3. Outputs

The routine returns a reference of a character string. If the data table is not empty, this character string corresponds to the key value of the data record immediately following the record whose key is referenced as the routine parameter. If the referenced record is not found or if it is the last in the table, a NULL pointer is returned.

2.4.5.4. Fault messages

None.

3. PACKAGE HTTP ANALYSIS

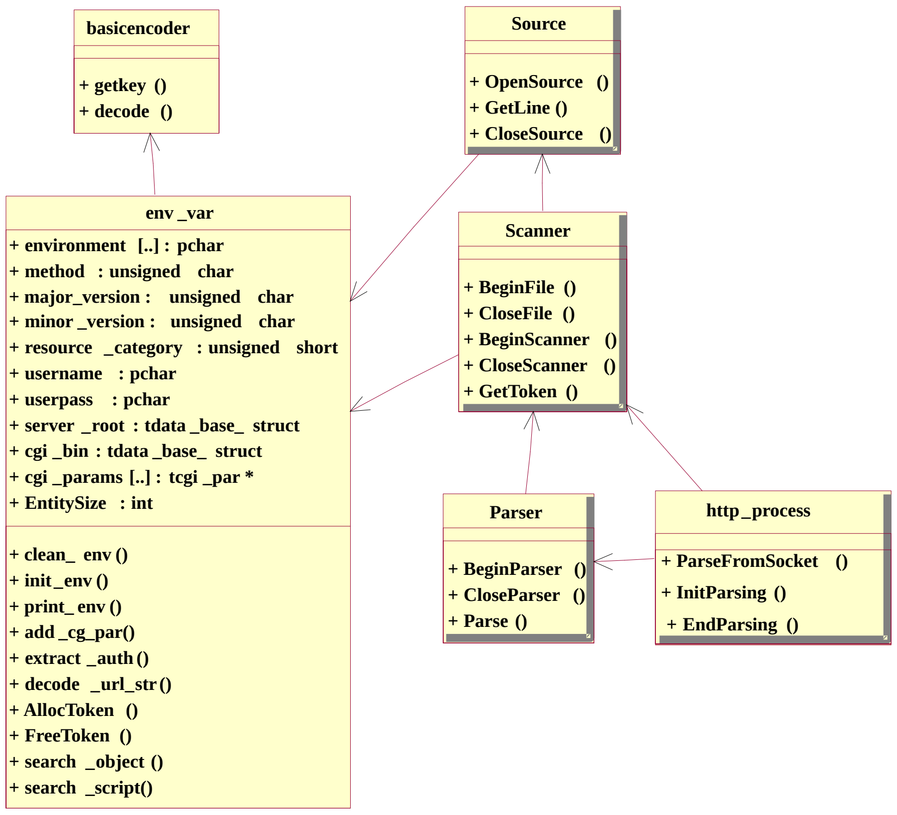


Figure 3.5 – Module relationships within the package *HTTP Analysis*

This package is designed in order to provide the support for the analysis of incoming request PDUs. It is composed of six modules:

● **http_process**	– provides the convenient wrap–up of PDU analysers;
● **Parser**	– implements the routine analysing the syntax of incoming PDUs; this module is constructed with the help of processors from the Cocktail toolbox [5];
● **Scanner**	– implements the routines which co–operate with the syntax analysers; the module is constructed with the help of processors from the Cocktail toolbox [5];
● **Source**	– implements the interface between the network and the syntax analyser; the interface of this module is inherited from the Cocktail toolbox, while the implementation is totally customised for the needs of this application;
● **env_var**	– implements the routines constructing the elements of the so called Common Gateway Environment (CGI); the routines are mostly used by the Parser module;
● **basicencoder**	– implements the decoding of “basic cookies” (base–64 coded name: password character chains) in certain kinds of request PDUs.

The user is expected to use only two of them: **`http_process`** (fully) and **`env_var`** (marginally). The modules: **`Parser, Scanner`** and **`Source`** are thought to be hidden by the **`http_process`** module. **`Scanner`** and **`Source`** modules are in principle not expected to be used directly. The advanced user may use the exposed interface of the **Parser** module.

Among two remaining modules, **`env_var`** is in principle considered as a collection of routines providing services to the **`Parser`** module. Some of its interface is used by the modules from other packages. The small module **`basicencoder`** is only used inside the package.

3.1. Module *http_process*

The module implements three basic operations linked to the analysis of incoming request PDUs: initialisation of CGI environment, active PDU parsing and finalisation of analysis. The routines implementing these operations are directly accessible to users who are not required to understand the details of parsing operations and environment construction routines. The routines exported by this module are used within the routines of the **`servengine`** package by default for the parsing and finalisation phases of the main server loop (see the two routines of § 2.1.)

3.1.1. Procedure *void InitParser(void)*

3.1.1.1. Description

This routine initialises the analysing process environment; it activates an isolated operation which prevents the scanning routine to crash in case of fatal errors. Although the fatal errors of parsers are extremely improbable, calling this routine is recommended at the beginning of the analysis phase of the server's loop (see § 2.1.). The fatal errors will cause the server to print on error stream a message:

Scanner Exception caught

3.1.1.2. Inputs

None.

3.1.1.3. Outputs

None.

3.1.1.4. Fault messages

None.

3.1.2. Procedure *unsigned short EndParsing(int socket_id)*

3.1.2.1. Description

This routine finalises the process of PDU analysis. It is proposed as the standard choice for the finalisation phase of the basic server loop (see § 2.1.).

3.1.2.2. Inputs

The secondary socket is to be closed by the routine.

3.1.2.3. Outputs

The result of a close operation. If this result is positive, the closing operation is successful.

3.1.2.4. Fault messages

None.

3.1.3. Procedure *unsigned short ParseFromSocket(int socket_id)*

3.1.3.1. Description

This routine organises the correct scheduling of operations during the phase of request PDU analysis. The application programmer can use this procedure without deep insight into the details of parser operation and environment construction. This procedure is proposed as a default solution for the phase of request analysis (see § 2.1.).

3.1.3.2. Inputs

The only input is the identifier of the server socket connected to the client. This socket receives the analysed PDU.

3.1.3.3. Outputs

The result returned by the routine reflects the status of the parsing process. If the parsing routine accepts the received PDU the result is 1; in the opposite case the result is 0.

3.1.3.4. Operational processing

The procedure chains the following operations:

- clean–up of environment variables,
- linking the active socket receiving the request to the input of the parsing routine,
- parser initialisation,
- parsing of the request with environment construction,
- parser termination,
- management of authentification.

The parsing process produces the following information, contained in the HTTP of the request PDU:

- requested method,
- protocol version (main version and release number),
- protocol options,
- requested resource name,
- requested resource class,
- authentification data (user name and password),
- request parameters,
- posted data (data sent via post service).

Not all the items listed above are found in each request.

The standard mode of operation for this routine provides the possibility of printing out on the standard output the results of environment synthesis as well as error messages detected by the parsing routines. This possibility can be eliminated by an appropriate choice of the compilation option. This option consists in defining the **`yySilent`** symbol during compilation of the module.

3.1.3.5. Fault messages

In its normal mode of operation the routine signals on its standard output the detection of syntax errors by the message:

Parser Error count = n

If n is different from 0, syntax errors are detected. If the compilation option enabling the printout of syntax errors is inhibited, no error messages are generated.

3.2. Syntax Analysing modules

One of the fundamental functions of the server software consists in analysing and extracting information from the incoming HTTP PDUs, which arrive via network in the form of character streams. The PDUs have a regular structure which can be described by a context free grammar. The formal description of this structure can be found in the official specification **[10]**.

The function of analysing character streams is a well known design pattern, implemented for example by the front–end modules of programming language compilers: scanners and parsers. Producing compiler front–end modules, although based on sophisticated theory, is now a well developed technology. It uses well defined algorithms easily mastered by the technique of automatic program generators, known as compiler compilers. The same technology can be adapted to the development of PDU analysing modules in this package.

The technique used here is based on the tools provided by Cocktail **[5]**– Compiler Compiler Toolbox Karlsruhe which is a collection of co–operating software tools aimed at automation of development of high efficiency compilers. The use of Cocktail tools makes it possible to derive the source code of some server software modules from their formal specification.

Any Cocktail based front–end is composed of a three stage processing process:

- structure of incoming PDU is analysed by **Parser** module which is automatically derived from PDU grammar following the specification from **[10]**;
- operation is facilitated by **Scanner** module which transforms the incoming character stream to a stream of tokens: (groups of characters forming composites such as separators, identifiers, numbers, etc.); the structure of tokens is also specified by a formal description;
- Scanner is adapted to source of character stream by **Source** module; the existence of this module makes the operation of Scanner and Parser independent of the origin of the character stream – this module, although furnished by Cocktail in its basic version, has to be implemented in this package.

The three modules listed above are furnished within this package and can even be used separately. In the normal context this does not make a lot of sense since they are designed to work together and not separately. Their co–operation is organised automatically by the processors of Cocktail. The full interface is relatively complicated although each module has a well identified natural entry point.

An advanced user can use the full interface to produce his own wrapper modules organising the PDU analysis process. For the majority of typical applications it is highly recommended not to use the modules directly but to activate them via the wrapper module **`http_process`** described in § 3.1.

3.2.1. Module *Source*

This module plays the role of an adapter operating between Scanner input and the source of the characters. In its standard application the module adapts the Scanner to a file system of the hosting machine. It handles such operations as opening and closing the files, reading characters, detecting end of file and handling errors. This makes the Scanner module independent and lets it operate at a high level of abstraction.

In the context of this application the generic version of the Source module furnished with the Cocktail toolbox has to be rewritten. The stream of characters in this case arrives from a network connection established via a socket interface, rather then from a disk file.

Despite this fact, the standard interface composed of three routines, is to be maintained in its original form in order to keep the mode of operation of Scanner and Parser modules unmodified. The interface functions are, in principle, never called directly by the software user. Here follows their description:

- **`int BeginSource(char*filename)`** – this routine redirects the Scanner input receiving the analysed character string from the standard input to a different source, in this case to an opened TCP/IP socket; the type of its input parameter, a character string signifying the stream symbolic name, is adapted to its original implementation; in this case the socket identifier (of int type), converted to character string (i.e. to char* type) is to be used; this parameter is passed from the user application to the routine indirectly, via the call of the Scanner interface function BeginFile; in the typical application even this call is not provided by the user who activates it by calling the wrapper routine ParseFromSocket;
- **`void CloseSource(int file_id)`** – this routine terminates the analysis from a character stream by closing it; it is never called directly from the application software; in a standard application the routine is activated from the call to EndParsing routine which calls the Scanner interface function CloseFile; the routine's input parameter has meaning only in the context of its calling environment i.e. inside the Scanner;
- **`int GetLine(int file_id,char* buffer, int buffsize)`** – this routine fills in the buffer provided by the analysers, with the characters provided by the input stream; the function is called deep inside the Scanner's structure and all its three input parameters have meaning only in the calling context of the Scanner module; the user only has the possibility of indirectly choosing buffer size (third parameter).

3.2.2. Module *Scanner*

This module is automatically derived from a formal specification by the use of the ***Rex*** processor **[6]**. The role of the Scanner consists in assembling the sequences of characters taken from the input stream into a series of ***tokens***, higher level entities which are used by the Parser. The Scanner specification is done in an appropriate language, which describes token structure in terms of ***regular expressions***. The equivalent compilable files in C language are produced automatically by ***Rex.*** The exported interface is relatively complicated; its role consists in supporting three phases of analysis operations: initialisation, parsing and termination. None of the interface elements are called directly by the user. The exhaustive description of the interface is provided in **[6]**.

3.2.3. Module *Parser*

This module is automatically derived from a formal specification by the use of the ***Lark*** processor **[7]**. The Parser module is the driving engine of the process of incoming PDU analysis. The Parser specification is done in an appropriate language, which describes structure of PDUs in terms of ***context free grammar***. The equivalent compilable files in C language are produced automatically by ***Lark.*** The exported interface is simple but its comprehension requires deeper insight into the theory of compilation. This interface is exhaustively described in **[7]**. The user of Field Device Web server software is freed from the burden of calling the Parser interface functions by the wrapper routine **`ParseFromSocket`**.

3.3. Module *env_var*

Request parameters needed to interpret correctly service requests addressed to the server are embedded within the structure of incoming PDUs. In order to structure the processing of these requests the server operation is split into the phases of parameter extraction and request execution. The parameter extraction phase is done during the parsing process. The parser routine calls the functions from the interface of this module in order to store the values of request parameters in the appropriate data structures, often called CGI parameters. Processing routines recover data structures from the created request context and execute programmed operations. The environment data structures are defined in the following table.

Exported variable	**Type or type of table elements**	**Description**
method	unsigned char	access method: POST or GET; possible values: httpGET =1 and httpPOST =2
major_version	unsigned char	version of HTTP protocol used (0 or 1)
minor_version	unsigned char	release of HTTP protocol used (9, 0, 1)
environment* [MaxEnv]	char*	table of strings which store the values of options extracted from the request (see table below)
resource_name[NLEN]	char	string holding the name of the requested resource; its maximum length is 80 in this software release
resource_type[NLEN]	char	string holding the extension (suffix) qualifying the requested object; in this implementation the following extensions are significant: `htm`, `html` for HTML pages; `gif, jpeg` for images; `class` for Java applets and `cgi` for scripts; lack of extensions qualifies the object as directory
resource category	unsigned char	numeric coding of object type; can take on values as follows: c_unknown=0, c_text = 1, c_gif=2, c_jpeg=3, c_java=4, c_dir =5, c_script =6
username	char*	string extracted from *authentification* option field corresponding to user name
userpass	char*	string extracted from *authentification* option field corresponding to user password
cgi_params[MAXPAR]	tcgi_par*	table of parameters (managed as stack) extracted from the header line of the request PDU
EntitySize	int	value extracted from the *Content/Length* option field; corresponds to length of entity appended to body of PDU in the case of POST method
server_root	tdata_base_struct	entry point to part of virtual file system containing pages, images and applets
cgi_bin	tdata_base_struct	entry point to part of virtual file system containing cgi scripts
resource_path	tdata_base_struct	path to object reference (see § 1.2.1.)

Table 3.6 – Environment data structures

The current version of the module is able to store up to thirty–two option values. The request analyser recognises and retrieves the values of nineteen types of option fields most frequently found in the request PDUs. These options are listed in the following table.

Value implemented	**Option type**	**Symbolic value implemented**
1	Referer	optReferer
2	Connection	optConnection
3	Host	optHost
4	User–Agent	optUserAgent
5	Forwarded	optForwarded
6	Accept	optAccept
7	Accept–Language	optAccepLang
8	Accept–Charset	optAcceptChar
9	UA–Pixels	optUAPixels
10	UA–Color	optUAColor
11	UA–OS	optUAOS
12	UA–CPU	optUACPU
13	Pragma	optPragma
14	Accept–Encoding	optAccepEnc
15	Content–Type	optConType
16	Content–Length	optConLen
17	authentification	optAuth
18	Via	optVia
19	Extension	optExtension

Table 3.7 – Common types of option fields

Option strings stored in the `environment` table can be used in the user–provided cgi scripts. For the non recognised options the retrieved values are stored in the 31st element of the table. In the case of many non recognised options in a request PDU, the element 31 stores the last option. Elements 0, 20–30 are not used and reserved for further extensions.

The module exports two types in order to support the retrieval of values of cgi parameters present in the header line of the request PDU:

- union adapted to store the values of one of the recognised types of cgi parameters (integer, float or string), defined as follows:

```
typedef union {
            pchar string;
            int integer;
            float real;
            }  url_val;
```

- structure storing values and types of cgi parameters, defined as follows:

```
typedef struct cgi_par_tag {
                    unsigned char par_type;
                    url_val value;
                    } tcgi_par;
```

Symbolic values corresponding to the recognised types of parameters are as follows:

c_string =0; c_int = 1; c_real=2.

On these data structures, some routines of this module are run and described in the following sections.

3.3.1. Function *char*AllocTokenStr(unsigned long length)*

3.3.1.1. Description

This routine is provided in order to create on the system heap a null–ended string. The string thus created can contain up to ***length*** characters. The string content is not initialised by this routine.

3.3.1.2. Inputs

Number of characters in the string (without terminating null character).

3.3.1.3. Outputs

Pointer to the reserved heap space.

3.3.1.4. Fault messages

If the allocation of heap space fails, the routine returns a NULL pointer.

3.3.2. Procedure *void FreeTokenStr(char* tokenstr)*

3.3.2.1. Description

The routine frees the heap frame occupied by the string allocated by the ***AllocTokenStr*** routine.

3.3.2.2. Inputs

Pointer to the heap frame to be freed.

3.3.2.3. Outputs

None.

3.3.2.4. Fault messages

None.

3.3.3. Procedure *void clean_env(void)*

3.3.3.1. Description

This routine empties all the significant environment variables, as follows:

- The heap space occupied by the strings stored in the ***environment*** table elements is freed and table elements are set to NULL,
- The content of the ***cgi_params*** table is emptied (space freed and elements set to NULL),
- username and userpass strings are freed and set to NULL,
- path object is released and its reference set to NULL,
- EntitiySize variable is set to 0,
- resource_category is set to c_unknown.

3.3.3.2. Inputs

None.

3.3.3.3. Outputs

None.

3.3.3.4. Fault messages

None.

3.3.4. Procedure *void init_env(void)*

3.3.4.1. Description

Simplified version of the routine *clean_env* which set the environment variables to default values without freeing the heap frames. This procedure is used mostly in the first phase of server operation.

3.3.4.2. Inputs

None.

3.3.4.3. Outputs

None.

3.3.4.4. Fault messages

None.

3.3.5. Procedure *void print_env(void)*

3.3.5.1. Description

This routine is used only for debugging purposes. It prints on the standard output (server console) the values of all environment variables in a comprehensive format.

3.3.5.2. Inputs

None.

3.3.5.3. Outputs

None.

3.3.5.4. Fault messages

None.

3.3.6. Procedure *void extract_auth(void)*

3.3.6.1. Description

This routine is used to extract the values of two environment variables: **`username`** and **`userpass`** from the string retrieved as the value of the **`authentification`** option in the **`environment`** table. The prerequisite for correct operation of this procedure is that this option be present in the request PDU and that the authentification coding is of the basic type (base–64 encoding). If these two conditions are not fulfilled, the routine fails and the result of its operation is unpredictable. In the case of successful operation the routine retrieves the encoded user name and user password strings and copies them to the previously allocated heap frames as null terminated strings. Heap frame allocation is performed using the *AllocTokenStr* routine.

3.3.6.2. Inputs

None.

3.3.6.3. Outputs

None.

3.3.6.4. Fault messages

None.

3.3.7. Procedure *void add_cgi_par(unsigned char par_type, url_val value)*

3.3.7.1. Description

This procedure is called in principle by the PDU parser. It pushes a new cgi parameter on the **`cgi_params`** stack if the stack is neither overflowing nor underflowing. The structures storing parameter types and parameter values are created on the system heap. If the parameter is of the **`c_string`** type, the corresponding string is first decoded (by the call to the **`decode_url_str`** routine) and then stored in a frame allocated in the system heap.

3.3.7.2. Inputs

The routine has two inputs: parameter type and parameter value.

3.3.7.3. Outputs

None.

3.3.7.4. Fault messages and failure modes

No explicit message is generated. In the case of stack overflow or underflow, the routine returns without any effective operation. If the creation of the heap frame for the structure fails, the routine returns with a NULL element pushed on the stack. In the case of the **`c_string`** type of parameter value, when the allocation of the heap frame fails, the value of the corresponding parameter pushed on the stack is unknown.

3.3.8. Function *char* decode_url_str(char* inpstr)*

3.3.8.1. Description

This routine implements the decoding of URL–encoded strings recovered as values of cgi parameters of type strings or as values of data of string types appended to POST service request PDUs. The routine, when successful, returns a decoded string corresponding to the input string containing URL–coded symbols. The decoding method implemented here is as follows:

- every ‘+’ symbol is replaced by a blank character,
- every group of three symbols composed of leading ‘%’ followed by a two–digit hexadecimal number is replaced by the ASCII character whose code corresponds to the hexadecimal number.

If the input string is NULL the routine returns NULL. In any other case the routine returns a string created on the system heap.

3.3.8.2. Inputs

The only input is a null terminated string extracted from the request PDU.

3.3.8.3. Outputs

Reference to a heap–allocated string containing the decoded version of the input string.

3.3.8.4. Fault messages

No explicit fault message is generated, however a NULL string is output when the input string is NULL or when heap frame allocation fails.

3.3.9. Function *tdata_base_struct search_object(void)*

3.3.9.1. Description

This routine implements the search for an object in the VFS within the tree containing passive server objects (i.e. directories, pages, images and applets). The routine has no input parameters and the search process is controlled by the values of environment variables: **`resource_path, resource_type, username`** and **`userpass`**. The search is run by the call to the **`FindNode`** routine (see § 1.3.7.) applied to the VFS tree having the root referenced by the **`server_root`** variable.

3.3.9.2. Inputs

None.

3.3.9.3. Outputs

Reference of the descriptor of the object searched for or a NULL pointer.

3.3.9.4. Fault messages

No explicit fault message is generated. Some failure modes are assimilated to the result of not finding the searched node. In both cases the output returned is NULL.

3.3.10. Function *tdata_base_struct search_script(void)*

3.3.10.1. Description

This routine implements the search for an object in the VFS within the tree containing active server objects (i.e. cgi scripts). The routine has no input parameters and the search process is controlled by the values of environment variables: **`resource_path, resource_type, username`** and **`userpass`**. The search is done by the call to the **`FindNode`** routine (see § 1.3.7.) applied to the VFS tree having the root referenced by the **`cgi_bin`** variable.

3.3.10.2. Inputs

None.

3.3.10.3. Outputs

Reference of the descriptor of the script object searched for or a NULL pointer.

3.3.10.4. Fault messages

No explicit fault message is generated.

3.4. Module *basicencoder*

This helper module implements the basic functions used to decode the incoming username/password pair coded according to a base–64 algorithm. The functions of this module are called uniquely within the context of the env_var module and serve to decode password and username variables arrived in the input PDU as a *base–64* coded string.

Base–64 coding is a process of replacing an ACSII character sentence by a coded character string according to the following procedure:

1. the original string is divided into a sequence of groups of three characters;
2. the group of three characters is considered to be an entity composed of 24 bits. The last group can have one , two or three characters. If the last group is incomplete, the corresponding entity is filled with 0 on its least significant bits;
3. Each entity is decomposed on 4 nibbles each having 6 bits. The nibbles are interpreted as integers; the values of these integers lie within the interval [0,63]. At that stage the string on n characters is replaced by the sequence of k integers where:

   k = ((n div 3) +m)*4 ;   if(n mod 3 ==0) m =0; else m=1;

4. The sequence of k integers is replaced by the sequence of k characters selected in the coding vector ***V*** composed of 64 ASCII characters. In the base–64 standard the coding vector is composed as follows:
   - first 26 places (0 – 25) are occupied by the uppercase letters **A–Z,**
   - next 26 places (26 – 51) are occupied by the lowercase letters **a–z**,
   - next 10 places (52 – 62) are occupied by **digits 0–9**,
   - last two places 62 and 63 are occupied by characters **+** and **/**
5. Each integer **x** is replaced by the character ***V[x]***; trailing zero integers are replaced by **=** character;
6. The algorithm produces the sequence of k ASCII characters.

The most widely implemented authentification procedure uses this coding algorithm to encrypt the user name and user password sent via the network. The two functions from this package are used to implement the reverse process i.e. the base–64 decoding algorithm.

3.4.1. Function *int getkey(char k)*

3.4.1.1. Description

This function is used to compute the place of a character provided as its input in the base–64 coding vector.

3.4.1.2. Inputs

The only input is the character to be decoded.

3.4.1.3. Outputs

The output corresponds to the place occupied by the character in the coding vector.

3.4.1.4. Fault messages

If the character does not belong to the coding vector, –1 is returned.

3.4.2. Function ***unsigned short decode_four(unsigned char inpbuf[4], char* outstr, unsigned char ptr)***

3.4.2.1. Description

This routine produces three ASCII characters corresponding to a 24–bits entity represented by the four elements of the vector passed via the first routine’s parameter. The ASCII characters are placed in the character string whose reference is passed via the second parameter. The characters are written into the string starting from the position indicated by the third routine’s parameter. The result returned corresponds to the next free place within the string, i.e. its value is equal to the value of the third parameter increased by three.

3.4.2.2. Inputs

The three input parameters are as follows:

- four–element array of integers containing 4 consecutive nibbles of 6 bits each which together form the 24–bit entity;
- pointer to an existing character string which is supposed to receive the decoded characters;
- offset with respect to the string’s beginning, starting from which the three retrieved characters are to be placed.

3.4.2.3. Outputs

The output corresponds to the first free place within the string from which the subsequent characters can be written; this value is equal to the value of the third input parameter increased by three.

3.4.2.4. Fault messages

No explicit failure mode is coded. The routine expects that the string referenced by the second parameter exists and that it has at least three places free in order to receive the decoded parameters. If this precondition is not fulfilled the result of the procedure operation is not predictable.

4. PACKAGE HTML GENERATION

The package regroups the modules which provide the services related to the on–line generation and handling of the embedded HTML pages.

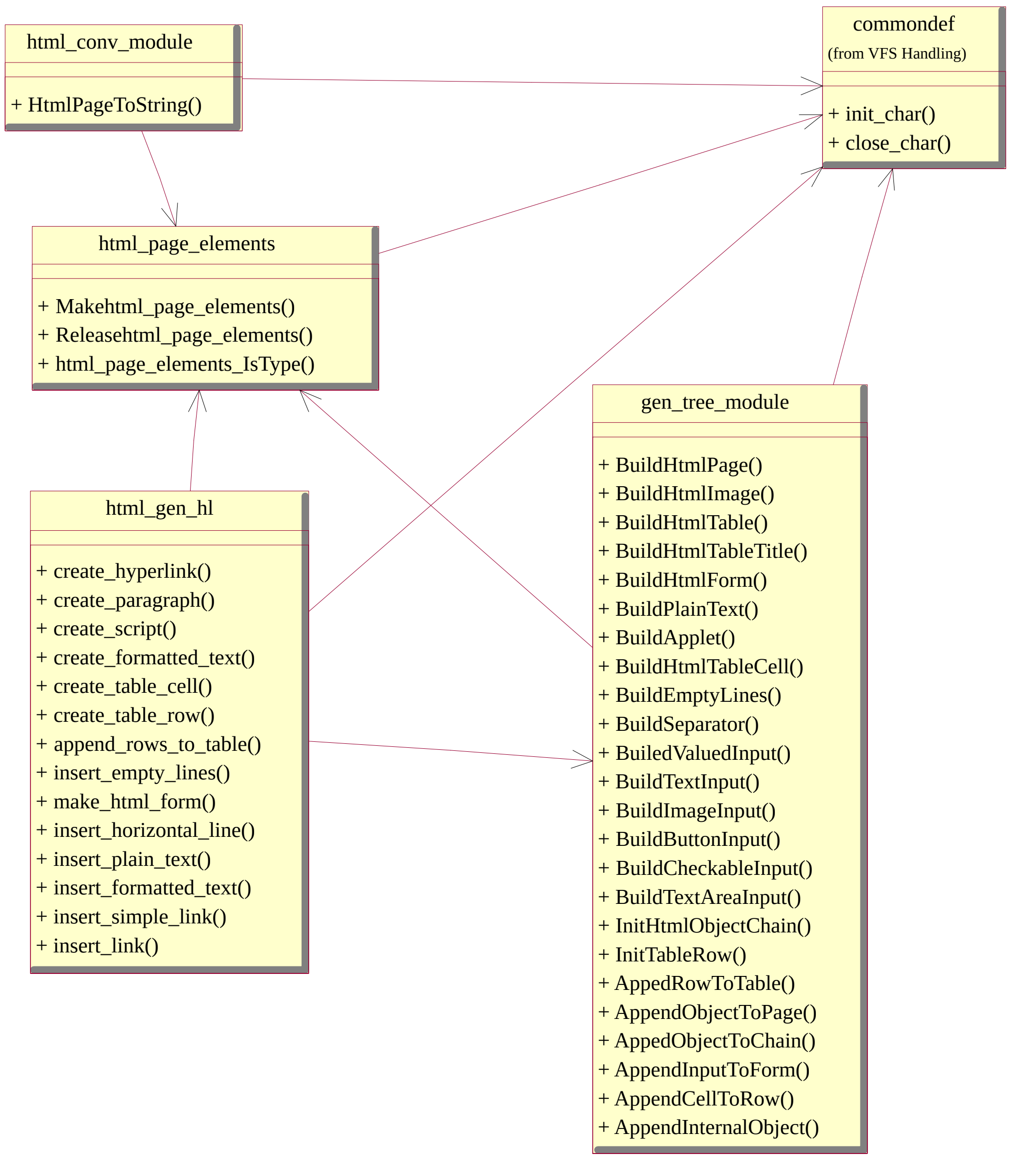


Figure 3.6 – Module relationship within the package *HTML GENERATION*

The package is composed of the four following modules:

● ***html_page_elements***	– defines the fundamental data type `thtml_page_elements` which is the basis of all embedded html page constructs;
● ***gen_tree_module***	– provides the routines supporting on–line generation of the embedded html page elements;
● ***html_gen_hl***	– completes the routines provided by the ***gen_tree_module*** by some higher level wrapper routines;
● ***html_cov_module***	– provides the front end routine converting the internal structure of embedded pages to a character oriented representation.

The relationships between the modules of the package are showed by the diagram in Figure 3.6.

4.1. Module *html_page_elements*

This module defines data types and routines on which is constructed the internal representation of embedded HTML pages. The most important item defined by the module is the data type `thtml_page_elements`. Although the module is coded in ANSI C programming language, only a small part is directly written in C. The vast majority of code, including the data type definition, is automatically generated by **ast**, one of the processors belonging to the Cocktail toolbox (see chap. 2 § 2). The original specification of the module is done using the **ast** syntax **[9]**.

Apart from the data type the module provides the definition of the three following functions:

4.1.1. Definition of thtml_page_elements data type

The definition of `thtml_page_elements` follows the regular scheme:

```
typedef union {
                    yobj_type_name_1 obj_type_name_1 ;
                    yobj_type_name_2 obj_type_name_2 ;
                    yobj_type_name_n obj_type_name_n ;
                    } * thtml_page_elements ;
```

The type is a pointer to a union which groups the structures. Each variant of the union designates the type of object belonging to the HTML objects belonging to the embedded server pages. These types have a very regular construction following this schematic:

```
typedef dll_obj_desc_tKind unsigned char;
                    typedef  { dll_obj_desc_tKind Kind ;
                    dll_obj_desc_tNodeHead yyHead ;
                         •
                         •
                     object attribute fields
                         •
                         •
                    } yobject name ;
```

The first field of each object is the tag `Kind` which identifies the object class. The second field of the record is used by the internal modules to manage the pool of objects.

The rest of each structure is composed of the series of object attribute definitions. In principle, the user must not modify these attributes manually and should rely on the set of routines furnished by this module as well as by the modules ***gen_tree_module*** and ***html_gen_hl***.

The type of the first structure field is the sub type of **`unsigned char`** and takes on the values from 1 to 35. The values of this type are redefined as a set of symbolic constants according to the following scheme:

Value	Symbolic name attributed	Kind of object designated
1	khtml_page	representation of an embedded HTML page
2	kobject_list	object class which represents collections of HTML objects contained in other structures
3	knoobjects	object marking the end of collection
4	khtml_objects	single element of object list
5	khtml_page_object	abstract class representing any element (tag) belonging to the HTML page, like paragraph, form, table, image, etc.
6	kparagraph	object representing paragraph tag
7	kscript	object representing script tag
8	kbasic_object	abstract class which represents any tag except paragraph and script
9	kinsertable_object	abstract class representing objects which can be inserted into other objects of basic_object type
10	kformatted_text	objects representing pieces of HTML text having such features as, colour, alignment style, font size etc.
11	khtml_image	object representing image embedding tags
12	kplaintext	objects representing plain non formatted text
13	kseparator	abstract class grouping empty lines and horizontal lines
14	kemptyline	object representing a group of empty line <BR> tags
15	kline	object representing horizontal lines
16	khyperlink	object representing hyperlink tag
17	khtml_table	object representing HTML table
18	khtml_form	object representing HTML form
19	kapplet	object representing applet embedding tag
20	ktable_cell	object representing a single cell of an HTML table
21	ktable_row	object representing a single row of an HTML table
22	ktable_title	object representing title of an HTML table
23	kinput_desc	abstract class representing tags which are used to build HTML forms
24	kinput	any form element except for an image
25	kbutton	button form field
26	ksimple_input	abstract class representing three types of form element classes: hidden, reset and submit
27	khidden	object representing form component of hidden type
28	ksubmit	object representing form component of submit type
29	kreset	object representing form component of reset type
30	ktext	object representing form component of one line text type
31	ktextarea	object representing form component of long text type
32	kcheckable	abstract class representing checkbox and radio button objects
33	kcheckbox	object representing a checkbox component
34	kradio	object representing a radio button component
35	kimage	object representing an image embedded within an HTML form

Table 3.8 – Symbolic name corresponding to the value of the first field

In the current version of embedded HTML there exist thirty–five types of objects in three groups:

- one high level class **`html_page`** representing complete HTML pages;
- one high level abstract class **`html_page_object`** representing any type of HTML tags which can be used with an embedded page;
- seven abstract classes representing types of HTML objects: **`basic_object, insertable_object, separator, input_desc, input, simple_input`** and **`checkable`**; these classes of objects are used very rarely, only when it is necessary to represent the common properties of other classes;
- three classes of HTML container objects: **`paragraph, html_table,`** and **`html_form`**; these objects represent the composite HTML objects which contain simple ones;
- nine classes of objects which represent active parts of an HTML form object: **`button, hidden, reset, submit, text, textarea, radio, checkbox`** and **`image`**,
- three classes of objects used to form collections (lists)**`:object_list, html_objects`** and **`noobject;`**
- three classes of intermediate level objects forming HTML tables**`: table_title, table_cell`** and **`table_row;`**
- eight classes of elementary objects representing the most frequently used HTML tags from which the embedded pages are constructed: **`script, formatted_text, plaintext, html_image, emptyline, line, applet and hyperlink.`**

Each class, whether abstract or concrete, has a number of attributes; the attributes for the non abstract classes are specified and described in the following table:

Object name	Attribute name	Attribute type	Description
html_page	title	pchar	character string containing html page title
	bgimage	pchar	character string containing background colour reference for the page
	bgname	pchar	character string containing background image reference for the page
	meta_refresh	unsigned char	value of meta refresh parameter in seconds
	meta_url	pchar	character string containing the url of object reference being the meta url parameter
	objects	thtml_page_elements	list of elements contained within the page body
paragraph	align	pchar	character string containing paragraph alignment style
	objects	thtml_page_elements	list of elements contained within the paragraph body
html_table	title	thtml_page_elements	pointer to object of title kind
	border	unsigned char	width of table border in pixels
	spacing	unsigned char	value of cell spacing parameter in pixels
	padding	unsigned char	value of cell padding parameter in pixels
	width	unsigned char	table width in %
	bgcolor	pchar	code of background color (i.e. # FFF000)

Table 3.9 – Description of non abstract class attributes

Object name	Attribute name	Attribute type	Description
	objects	thtml_page_elements	list of row objects
html_form	method	pchar	character string containing name of method applied when form is submitted (POST or GET)
	action	pchar	character string containing the url of the object activated on the server as the result of form submission
	objects	thtml_page_elements	list of objects contained with the form
html_objects	object	thtml_page_elements	pointer to the object in the list head
	next	thtml_page_elements	pointer to the rest (tail) of the list
noobject	——————— –	———————————	No parameters
table_title	align	pchar	character string containing title's alignment style
	tobj:	thtml_page_elements	list of objects included in the title
table_cell	align	pchar	character string containing alignment style
	width	unsigned char	relative cell width in %
	colspan	unsigned char	horizontal cell span in number of columns
	rowspan	unsigned char	vertical cell span in number of rows
	objects	thtml_page_elements	list of objects contained in the cell
table_row	objects	thtml_page_elements	list of cell objects in the row
script	language	pchar	character chain containing script language name
	objects	thtml_page_elements	list of plain text paragraphs contained in the script
formatted_text	color	pchar	character string containing font color code
	size	unsigned char	font size in points
	isbold	unsigned char	flag set to a non zero value when font is bold
	isblinking	unsigned char	flag set to a non zero value when font is blinking
	style	unsigned char	value of *1* to *n* for styles *H1* to *Hn*
	objects	thtml_page_elements	pointer to the list of objects contained within this object
plaintext	text	pchar	pointer to the chain of characters containing the text
html_image	src	pchar	character string containing url of image source file
	alt	pchar	character string containing the "alternative" string
	width	unsigned char	image maximum width in pixels
	height	unsigned char	image maximum height in pixels
	border	unsigned char	image maximum border width in pixels
emptyline	nr	unsigned char	number of consecutive empty lines within the page generated by this object
line	width	unsigned char	relative line width in %
	size	unsigned char	line thickness in points
	align	pchar	character string containing line's align style
applet	code	pchar	character string containing the url of applet's source file
	codebase	pchar	character string containing the url of the directory containing applet's source file

Table 3.9 – Description of non abstract class attributes (continued)

Object name	Attribute name	Attribute type	Description
applet	alt	pchar	character string containing the “alternative” string
	width	unsigned char	width of area reserved for applet in pixels
	height	unsigned char	height of area reserved for applet in pixels
hyperlink	ref	pchar	string of characters containing url of referenced object of the link
	target	pchar	string of characters containing reference of browser’s object in which the link reference is to be displayed
	isref	unsigned char	flag set to a non zero value when the link is a hyperlink, and set to zero when it is an anchor
	objects	thtml_page_elements	pointer to the list of objects contained within the hyperlink
button	name	pchar	string of characters containing the object’s name
	value	pchar	string of characters which will be displayed on the button
	onClick	pchar	string of characters containing the name of script activated by the click on the button
hidden	name	pchar	string of characters containing the object’s name
	value	pchar	string of characters attributed as the object’s value
reset	name	pchar	string of characters containing the object’s name
	value	pchar	string of characters which will be displayed on the reset button
submit	name	pchar	string of characters containing the object’s name
	value	pchar	string of characters which will be displayed on the submit button
text	name	pchar	string of characters containing the object’s name
	value	pchar	string of characters displayed in the object at page activation
	size	unsigned char	height of the object
	maxlength	unsigned char	maximum length of the object
textarea	name	pchar	string of characters containing the object’s name
	value	pchar	string of characters displayed in the object at page activation
	rows	unsigned char	height of object in rows
	cols	unsigned char	width of objects in columns
radio	name	pchar	string of characters containing name of radio button group
	value	pchar	string of characters which gives the name of the particular element of the group
	checked	unsigned char	check flag, when set to a non zero value causes the display of the button as checked
checkbox	name	pchar	string of characters containing name of checkbox group
	value	pchar	string of characters which gives the name of the particular element of the group
			check flag, when set to a non zero value causes the display of the checkbox as checked
image	name	pchar	string of characters containing the object’s name
	src	pchar	string of characters containing url of file which stores the image

Table 3.9 – Description of non abstract class attributes

4.1.2. Function *thtml_page_elements Makehtml_page_elements(uchar Kind)*

4.1.2.1. Description

This routine creates an object of the type specified in Table 3.9 and returns the reference to it. The object is created in the system heap.

4.1.2.2. Inputs

The input parameter specifies the kind of the object to be created; its value should correspond to one of the constants specified in the Table 3.8.

4.1.2.3. Outputs

The returned value is a reference of the newly created object (pointer to the object) if creation succeeds.

4.1.2.4. Fault messages

In case of failure the procedure returns the NULL pointer.

4.1.3. Procedure *void Releasehtml_page_elements(thtml_page_elements object)*

4.1.3.1. Description

This routine destroys the object referenced by the input parameter. The heap area occupied by the object is returned to the system.

4.1.3.2. Inputs

The input parameter references the object to be destroyed.

4.1.3.3. Outputs

None.

4.1.3.4. Fault messages

None.

4.1.4. Function *html_page_elements_IsType(thtml_page_elements object, uchar Kind)*

4.1.4.1. Description

This routine checks whether the object referenced by its first parameter is of the type referenced by the second parameter. The object is considered to be of the given type in two cases:

- it belongs to a sub type of the specified type: ex: **`plaintext`** is of type **`kbasic_object`** and **`kinsertable_object;`**
- it is strictly of the specified type **`applet`** or is of type **`kapplet`**.

4.1.4.2. Inputs

The first input parameter references the object, the second provides the type to be checked with.

4.1.4.3. Outputs

The function returns 1 if the object is of the specified type and 0 if it is not.

4.1.4.4. Fault messages

None.

4.2. Module *html_gen_hl*

This module groups routines of two types. The first group is composed of eight routines which create *container objects* of the **`thtml_page_elements`** type. Container objects are objects which possess a collection of other objects. Tables, scripts, links, and formatted texts are container objects since they can contain *simple objects* like plain text paragraphs, images, or separator lines. They can also contain other containers: a table can contain a link or a form.

The second group is composed of six routines whose role consists in inserting certain objects of the **`thtml_page_elements`** type in the **`html_page`** object.

The functional similarity between the routines in each group allows us to give their collective, concise description as provided in the following sections.

4.2.1. Routines generating container objects

The operations of all the routines which belong to this group are similar. They consist of two successive operations:

1. creation of an empty container object featuring the characteristics described by its attributes passed via routine parameters;
2. filling in of the container with the collection of objects passed via routine parameters.

The signatures of the routines follow a regular scheme:

thtml_page_elements *<routine name>*

(*<attribute params section>****, unsigned char nr_of_objects,...)***

One can distinguish three groups of parameters:

- variable number of parameters *`<attribute params section>`* conveying the values of container object attributes (object dependent number, order and attribute type);
- parameter **`nr_of_objects`** conveying the number of objects to be inserted in the container (one parameter);
- n parameters, represented in the scheme by **`...,`** conveying the references of objects inserted in the container.

All the routines return the objects of the **`thtml_page_elements`** type.

4.2.1.1. Description

The following table contains the description of the routines.

Routine	Description
create_hyperlink	creates a hyperlink object and fills it in with a set of n objects passed by the last n routine parameters
create_paragraph	creates a paragraph object and fills it in with a set of n objects passed by the last n routine parameters
create_script	creates a script object and fills it in with a set of n objects passed by the last n routine parameters
create_formatted_text	creates a formatted text object and fills it in with a set of n objects passed by the last n routine parameters
make_html_form	creates a form object and inserts into it a set of n objects passed by the last n routine parameters
create_table_cell	creates a table_cell object and inserts into it a set of n objects passed by the last n routine parameters
create_table_row (uchar : , ... :)	creates a table_row object and inserts into it a set of n table_cell objects passed by the last n routine parameters
append_rows_to_table	inserts into an existing, empty html table object a collection of n rows passed by n last parameters

Table 3.10 – Routine description

4.2.1.2. Inputs

The signature of the functions from the first group reveals the regularity of the parameters mentioned in § 4.2.1. The semantics of the parameters belonging to the second and third group (`unsigned char nr_of_obj` and `...`) is straightforward and common to all routines. On the contrary, the parameters of the first group vary in number and in significance from one routine to another. For this reason they require a more detailed description, contained in the following table.

Routine	Parameter	Parameter types	Description
create_hyperlink (pchar ref, pchar target, unsigned char isref, unsigned char nr_of_obj, ...)			
	ref	pchar	URL of objects referenced by the link
	target	pchar	target for displaying object referenced by ref attribute
	isref	unsigned char	flag which when set signifies that the object is a hyperlink and when reset means that the object is an anchor
create_paragraph (pchar align,unsigned char nr_of_obj, ...)			
	align	pchar	alignment style of the paragraph
	language	pchar	
create_formatted_text (pchar colour, unsigned char size, unsigned char bold, unsigned char blink, unsigned char style, unsigned char nr_of_obj, ... :)			
	colour	pchar	font colour denomination
	size	unsigned char	font size

Routine	Parameter	Parameter types	Description
	bold	unsigned char	flag setting the bold font
	blink	unsigned char	flag setting the blink mode of font
	style	unsigned char	number of style defined by HTML (H1–H5)
make_html_form (pchar action, pchar method, unsigned char nr_of_obj, ...)			
	action	pchar	URL of an object which will be activated after the form submission
	method	pchar	method used for the submission (POST or GET)
create_table_cell pchar align, unsigned char width, unsigned char collspan unsigned char rowspan, uchar char nr_of_obj,.. . .)			
	align	pchar	alignment style of the cell
	width	unsigned char	cell width in %
	collspan	unsigned char	cell column span in pixels
	rowspan	unsigned char	cell row span value in pixels
create_table_row (unsigned char nr_of_objects)			
No parameters in the first group since row object does not have any particular attribute			
append_rows_to_table (thtml_page_elements table: , unsigned char nr_of_objects,…)			
	table	thtml_page_elements table	pointer to a table to which rows will be appended

Table 3.11 – Description of first group parameters

4.2.1.3. Outputs

Each function generates an object referenced by the pointer of the `thtml_page_elements` type. The kind of object corresponds to the routine's role and is described in the previous table.

4.2.1.4. Fault messages

No fault message is generated. If a routine fails to create an appropriate object, the NULL pointer is returned.

4.2.2. Routines which append simple objects to html pages

4.2.2.1. Description

The role of these routines consists in appending an html object to a page object. The module provides six routines whose role is described in the following table.

Routine	Role
insert_empty_lines	Appends a series of empty lines to the page
insert_horizontal_line	Appends a horizontal line to the page
insert_plain_text	Appends a plain text paragraph to the page
insert_formatted_text	Appends a formatted text object to the page
insert_link	Appends an html link to the page
insert_simple_link	Appends a simplified link (with plain text only) to the page

Table 3.12 – Routine role

4.2.2.2. Inputs

The signatures of the routines follow a regular scheme:

thtml_page_elements *<routine name>* ***(thtml_page_elements page ,****<attribute section>****)***

The first input parameter of the procedure is always the reference of an HTML page object on which the routine will operate. This parameter is followed by the object attribute section whose contents varies from one routine to the other. The following table describes the attribute section for each routine of the group.

Routine	Parameter	Meaning
insert_empty_lines (thtml_page_elements page, unsigned char nr_of_lines)		
	unsigned char nr_of_lines	number of empty lines to insert
insert_horizontal_line (thtml_page_elements page, unsigned char width, unsigned char height, pchar align)		
	unsigned char width	width of the line in %
	unsigned char height	height of the line in points
	pchar align	reference of a string representing line’s alignment mode
insert_plain_text (thtml_page_elements page, pchar text)		
	pchar text	reference of the text to be inserted
insert_formatted_text (thtml_page_elements page, pchar text, pchar colour, unsigned char size, unsigned char, unsigned char, unsigned char)		
	pchar *text	reference of the contained text of the *formatted_text* object
	pchar colour	denomination of the font colour
	unsigned char size	font size
	unsigned char is Bold	when set, this flag formats the text to bold
	unsigned char is Blinking	when set, this parameter makes the text blink
	unsigned char style	paragraph style (predefined from H1 to H5)
insert_link (thtml_page_elements page, thtml_page_elements element_list, pchar ref, pchar target, unsigned char isref)		
	thtml_page_elements element_list	list of elements contained in the link object
	pchar ref	URL of objects referenced by the link
	pchar target	target for displaying the object referenced by ref attribute
	unsigned char isref	when set, flag signifies that the object is a hyperlink and when reset, means that the object is an anchor
insert_simple_link (thtml_page_elements page, pchar text, pchar ref, pchar target, unsigned char isref)		
	pchar	reference of text to be inserted into link
	pchar ref	URL of objects referenced by the link
	pchar target	target for displaying the object referenced by ref attribute
	unsigned char isref	flag which when set signifies that the object is a hyperlink and when reset means that the object is an anchor

Table 3.13 – Description of the attribute section for the six first group routines

4.2.2.3. Outputs

All the routines from the group return a reference to an object of the **`thtml_page_elements`** type. The result is normally the pointer to the page with the appended object.

4.2.2.4. Fault messages

No fault message is generated. If the operation programmed by the routine fails, it returns a NULL pointer.

4.3. Module *gen_tree_module*

This module contains the routines which realise two types of functions:

- creation of HTML related objects;
- extension and modification of HTML related objects.

They are described in the following sections:

4.3.1. Routines which generate HTML related objects

The routines from this group are specialised in generation of an object of a given type. Usually the name of the routine is significant and tells which type of object the routine creates. The following list introduces the routines and their signatures, i.e. the list of parameter types between brackets.

- BuildHtmlPage (pchar, pchar, pchar, unsigned char, pchar, thtml_page_elements);
- BuildHtmlImage (pchar, pchar, unsigned char, unsigned char, unsigned char);
- BuildHtmlTable (thtml_page_elements, thtml_page_elements, unsigned char, unsigned char, unsigned char, unsigned char, pchar);
- BuildHtmlTableTitle (thtml_page_elements:, pchar:);
- BuildHtmlForm (thtml_page_elements:, pchar:, pchar:);
- BuildPlainText (pchar);
- BuildApplet (pchar, pchar:, pchar, unsigned char, unsigned char);
- BuildHtmlTableCell (pchar, unsigned char, unsigned char, unsigned char, thtml_page_elements:);
- BuildEmptyLines (unsigned char);
- BuildSeparator (unsigned char, unsigned char, pchar);
- BuiledValuedInput (unsigned char, pchar, pchar);
- BuildTextInput (pchar, pchar, unsigned char, unsigned char);
- BuildImageInput (pchar, pchar);
- BuildButtonInput (pchar, pchar, pchar:);
- BuildCheckableInput (unsigned char, pchar, unsigned char);
- BuildTextAreaInput (pchar, unsigned char, unsigned char, pchar).

4.3.1.1. Description

Each routine, when successful, generates an object of an appropriate kind on the system heap. To do this the routines use the function `Makehtml_page_elements` from the module **html_page_elements.**

4.3.1.2. Inputs

Input parameters of the functions convey the values of object attributes. Their number and type depend on the kind of objects constructed by the function. The following table describes the parameters of each group function.

Function name	Parameters		
	name	**type**	**description**
BuildHtmlPage	title	pchar	character string containing page title
	bg_image	pchar	character string containing URL of background image
	bg_colour	pchar	background color code
	meta_refresh	unsigned char	value of refresh period in seconds
	meta_url	pchar	character string containing url of replacement page
	obj_list	thtml_page_elements	pointer to the list of objects contained within the page
BuildHtmlImage	source	pchar	character string containing URL designating location of (virtual) file which stores the image
	alt	pchar	character string which is displayed when image is pointed to by cursor
	width	unsigned char	width of the image
	height	unsigned char	height of the image in pixels
	border	unsigned char	width of the image border in pixels
BuildHtmlTable	objects	thtml_page_elements	pointer to the list of table rows
	title	thtml_page_elements	pointer to the table title object
	border	unsigned char	width of table’s border in pixels
	spacing	unsigned char	cell spacing in pixels
	padding	unsigned char	cell padding in pixels
	width	unsigned char	relative table width in %
	bg_color	pchar	background color code
BuildHtmlTableTitle	objects	thtml_page_elements	pointer to the list of objects contained in the title (images, plain text or formatted text objects)
	align	pchar	title alignment description
BuildHtmlForm	objects	thtml_page_elements	pointer to the list of objects contained in the form
	method	pchar	character string designating method used on the form submission
	action	pchar	character string designating the server object activated by the form submission
BuildPlainText	text	pchar	pointer to the character string containing the text
BuildApplet	code	pchar,	name of virtual file containing applet bytecode
	codebase	pchar:,	virtual directory containing applet code
	alt	pchar,	character string which is displayed when image is pointed to by the applet
	width	unsigned char	width of the applet in pixels
	height	unsigned char	height of the applet in pixels
BuildHtmlTableCell	align	pchar,	alignment code for objects contained wthin the cell
	width	unsigned char	cell width in % of table width
	colspan	unsigned char	nb of columns which span the cell
	rowspan	unsigned char	nb of rows which the cell spans
	objects	thtml_page_elements	list of objects contained within the cell

Table 3.14 – Description of parameters according to function

Function name	**Parameters**		
	name	**type**	**description**
BuildEmptyLines	nr_of lines	unsigned char	nb of empty lines represented by the object
BuildSeparator	width	unsigned char	width of the line in %
	height	unsigned char	thickness of the line in pixels
	align	pchar	alignment code for the line
BuiledValuedInput	kind	unsigned char	this parameter indicates the kind of object; it can take only predefined values which are: **`khidden,`** **`ksubmit`** or **`kreset`**
	name	pchar	character string which contains the input identifier
	init_value	pchar	value given to the object; semantics of this parameter vary from one type of object to another
BuildTextInput	name	pchar	character string which contains the input identifier
	value	pchar	text which is initially displayed in the input
	size	unsigned char	height of the input in pixels
	maxlength	unsigned char	length of the input given in number of characters.
BuildImageInput	name	pchar	character string which contains the input identifier
	source	pchar	path to the (virtual) file which contains the image contents
BuildButtonInput	name	pchar	character string which contains the button identifier
	value	pchar	character string which contains the text which is to be displayed on the button
	onClick	pchar	character string which contains the name of the script activated when the button is clicked
BuildCheckableInput	kind	unsigned char	this parameter indicates the kind of object; it can take on only predefined values which are: **`kradio`** or **`kcheckbox`**
BuildCheckableInput	name	unsigned char	character string which contains identifier of group to which object belongs
	value	pchar	character string which contains the name of the object
	checked	unsigned char	flag which indicates if the object is displayed as checked
BuildTextAreaInput	name	pchar	character string which contains the input identifier
	rows	unsigned char	control's height in rows
	cols	unsigned char	control's width in columns
	text	pchar	text to be displayed initially in the control

Table 3.14 – Description of parameters according to function (continued)

4.3.1.3. Outputs

All the routines return a pointer of the **`thtml_page_elements`** type which references the created object of a given kind. In case of failure of the routine, a NULL pointer is returned.

4.3.1.4. Fault messages

There is no explicit fault message. The only reason for a routine to fail is lack of room in the system heap. In this case a NULL pointer is returned.

4.3.2. Routines which modify the structure of HTML related objects

This set of routines is designed to complete the generation of objects such as pages, forms, table rows and tables. All these objects can contain other objects normally attached to them as a chain of objects. The following routines help to create and handle the chains.

4.3.2.1. Description

This group of functions is composed of seven routines described in the following table.

Routine	Parameters	Description
InitHtmlObjectChain	thtml_page_elements firstobj	this routine initialises a list of objects and attaches the pointer passed to it by the first parameter
AppedObjectToChain	thtml_page_elements chain thtml_page_elements object	this routine attaches the object pointed to by its second parameter to the chain pointed to by the second parameter
AppendInternalObject	thtml_page_elements int_obj thtml_page_elements object	this routine attaches the object pointed to by its second parameter to a container object pointed to by the first parameter; container objects admitted in this routine are: *paragraph, formatted text hyperlink, script and table cell*
AppendObjectToPage	thtml_page_elements page thtml_page_elements object	this routine attaches the object pointed to by its second parameter to an HTML page object specified by the first parameter
AppedRowToTable	thtml_page_elements table thtml_page_elements row	this routine attaches the table row object pointed to by its second parameter to an HTML table object specified by the first parameter
AppendCellToRow	thtml_page elements row thtml_page_elements cell	this routine attaches the table cell object pointed to by its second parameter to a table row object specified by the first parameter
AppendInputToForm	thtml_page_elements form thtml_page_elements input	this routine attaches the form input object pointed to by its second parameter to an HTML form object specified by the first parameter

Table 3.15 – Description of routines

4.3.2.2. Inputs

The routine signature has a very regular structure, as it can be seen from the previous table. The meaning of both parameter functions is always the same: the first means container object and the second means the object to be added to the list of contained objects.

Only first routine is different from the others since its role consists in preparation of an object chain for further usage.

4.3.2.3. Outputs

thtml_page_elements –pointer to an object of a given type.

4.3.2.4. Fault messages

None.

4.4. Module *html_conv_module*

The role of this module consists in transformation of HTML related objects to character strings ready to be transported through the network. The module contains many local routines, each specialised in transformation of a particular type of object. The only routine accessible from the outside of the module is the **`HtmlPageToString`** function described in the following section.

4.4.1. Function *unsigned short HtmlPageToString (pchar buffer, unsigned char buff_ptr, thtml_page_elements html_page_obj)*

4.4.1.1. Description

This function converts a complete, well formed HTML page object into a zero–ended string of printable characters. The string is placed in a buffer and can be either processed locally in order to obtain page modifications or can be sent via the network to an http client (Internet browser). The routine design is based on the principle that a large enough character buffer is available for the reception of the generated character string.

4.4.1.2. Inputs

The routine has three inputs:

- reference of the memory area (character buffer) which will store the string created by the routine;
- initial offset with respect to the beginning of the area from which the string will be written into the buffer;
- reference of the HTML page object to be converted.

4.4.1.3. Outputs

The routine returns the offset of the first byte immediately after the end of the string, written in the buffer i.e. the first byte free in the buffer.

4.4.1.4. Fault messages

No error message is generated explicitly. When the routine fails to convert the object, the value returned is equal to the value of the second input parameter.

Appendix A — HTTP protocol specification for Field Device Web Server

This section contains the specification of the subset of the HTTP 1.0 protocol recognised by the analysing package described in chapter 3 § 3. The most important restrictions with respect to the full specification, described in **[10]** are as follows:

1. Only POST and GET services are recognised;
2. The object to be served is described by its access path. NULL access path describes server "by default object" which should be on the server;
3. The access path syntax is as follows:

   /ident_step_1/ident_step_2/../ident_step_n/object_id[.ext]

4. The type of requested object is identified uniquely by its extension. The extensions recognised by this software are:

`htm, html`	HTML page
`cgi`	CGI script
`gif, jpeg`	image
`class`	applet
no extension	directory

5. The only legal characters in the path elements (identifiers of directories) are upper– and lowercase letters, figures, underscore, minus and plus;
6. Identifiers in URL–encoded form content can additionally contain periods;
7. The identifiers should be shorter than 80 characters.

1. CONTEXT–FREE GRAMMAR OF HTTP REQUEST

The text which follows represent the listing of the context free grammar describing the operation of the Parser module (see chap. 3 § 3.2.3.) generated by the Lark processor **[7]** from the Cocktail toolbox.

```
PARSER
EXPORT {
#ifdef __cplusplus
  #include ”env_class.h”
#endif
GLOBAL {
#include <stdlib.h>
# define yyInitStackSize 20
# define NO_RECOVER
#define ERROR printf(”Syntax error in request\n”);
#include ”commondef.h”
#ifdef __cplusplus
#define EnvObjectPrefix http_env.
#else
  #include ”env_var.h”
  #define EnvObjectPrefix
#endif
  #include ”data_base_module.h”

unsigned short
recover_res_category(const char * res_ext)
{
   if (strcmp(res_ext, ”htm”) == 0 ||
       strcmp(res_ext, ”html”) == 0 ||
       strcmp(res_ext, ”HTM”) == 0 ||
       strcmp(res_ext, ”HTML”) == 0)
          return c_text;
   else if (strcmp(res_ext, ”gif”) == 0 ||
           strcmp(res_ext, ”GIF”) == 0)
          return c_gif;
   else if (strcmp(res_ext, ”jpeg”) == 0 ||
           strcmp(res_ext, ”JPEG”) == 0)
          return c_jpeg;
   else if (strcmp(res_ext, ”class”) == 0 ||
           strcmp(res_ext, ”CLASS”) == 0)
          return c_java;
   else if (strcmp(res_ext, ”cgi”) == 0 ||
           strcmp(res_ext, ”CGI”) == 0)
          return c_script;
   else
          return c_unknown;
}
}
PROPERTY INPUT
RULE
http_request = <
   =request_header END_MAIN .
   =request_header  opt_list END_MAIN  .
   =request_header  opt_list END_MAIN urlcoding .
 > .
request_header =<
   = method:ORDER uri version
          {=>{

          EnvObjectPrefix method = method:Value;

          };
          } .
   = method:ORDER uri URLSEP urlcoding version
          {=>{
          EnvObjectPrefix \method = method:Value;
          };
          } .
          = method:ORDER uri  ’–’ cgi_params URLSEP urlcoding version
          {=>{
          EnvObjectPrefix \method = method:Value;
          };
          } .
   = method:ORDER uri  ’–’ cgi_params version
          {=>{
          EnvObjectPrefix \method = method:Value;
          };
          } .
>.

version = HTTP SLASH v:FLOAT
```

```
          { =>{

               EnvObjectPrefix major_version = v:Value;
               EnvObjectPrefix minor_version = (v:Value*10-10);

              };
          } .
       uri = <
           = SLASH      { =>{

                            strcpy(EnvObjectPrefix resource_name,ROOT);
                            EnvObjectPrefix resource_category = c_text;
                            strcpy(EnvObjectPrefix resource_type,ROOT);
                            };
                        } .

            = path     { =>{

                              strcpy(EnvObjectPrefix resource_type,"dir");
                              EnvObjectPrefix resource_category =c_dir;
                              EnvObjectPrefix recover_res_name();

                            };
                        }.

           = path POINT ext:IDENT { =>{

                                      strcpy(EnvObjectPrefix resource_type,ext:Ident);
                                      EnvObjectPrefix resource_category =
       recover_res_category(ext:Ident);
                                      EnvObjectPrefix recover_res_name();

                                      FreeTokenStr(ext:Ident);
                                     };
                                 }.
           = path SLASH { =>{

                             EnvObjectPrefix recover_res_name();
```

```
                              EnvObjectPrefix resource_category =c_dir;
                              strcpy(EnvObjectPrefix resource_type,”dir”);

                           };
                        }.
            > .



        cgi_params = <
                  = cgi_param  .
                  = cgi_params ’+’ cgi_param .
        > .
        cgi_params_list = <
                  = cgi_param  .
                  = cgi_params ’,’ cgi_param .
        > .
        cgi_param = <
                  = id:IDENT    { =>{ url_val trpar;
                                      trpar.string=id:Ident;

                                     (void)EnvObjectPrefix add_cgi_par(c_string,trpar);
                                     FreeTokenStr(id:Ident);
                                    };
                                } .
                  = nr:NUMBER   { =>{ url_val trpar;
                                      trpar.integer =nr:Value;
                                     (void)EnvObjectPrefix add_cgi_par(c_int,trpar);
                                    };
                                } .
                  = nr:FLOAT    { =>{ url_val trpar;
                                      trpar.real =nr:Value;

                                     (void)EnvObjectPrefix add_cgi_par(c_float,trpar);

                                    };
                                } .
        > .
```

```
step = SLASH name:IDENT {=>{
                      EnvObjectPrefix resource_path = AppendStep(EnvObjectPrefix
resource_path,name:Ident);
                      FreeTokenStr(name:Ident);


                       };
                  }.
path = <
     = step .
     = path step  .
     >.
 opt_list =<
        = option .
        = opt_list  option   .
> .
  option = <
        =code:OPTION val: RECUP { =>{
                                           if (EnvObjectPrefix
environment[code:Value] !=NULL)


                                        FreeTokenStr(EnvObjectPrefix
environment[code:Value]);
                                    EnvObjectPrefix environment[code:Value]
=val:String;


                                     if(code:Value ==optContLen)
                                         EntitySize =EntitySize+atoi(EnvObjectPrefix
environment[code:Value]);
                                      };
                                  }.
        =alien:IDENT val: RECUP   {=>{ /*free(alien:Ident);*/
                                     FreeTokenStr(alien:Ident);
                                    /* free(val:String);*/
                                     FreeTokenStr(val:String);
                                    };}.
>.
urlcoding = <
          = cgi_params_list .
          = assign .
          = urlcoding SEP assign .
```

```
> .
 assign          =<
         = name:IDENT ASSIGN rhs
         {=>{  void* lvaladdr;

             if (rhs:type ==c_string && rhs:Name)
                      lvaladdr     =  &rhs:Name;
                 else if (rhs:type ==c_int)
                         lvaladdr     =  &rhs:IValue;
                        else lvaladdr     =  &rhs:RValue;
            (void)set_db_data(name:Ident,rhs:type,lvaladdr);

            /*free(name:Ident);*/
            FreeTokenStr(name:Ident);
            if (rhs:type ==c_string && rhs:Name)
                /*free(rhs:Name);*/
                FreeTokenStr(rhs:Name);
                };
            } .
            = alien:IDENT ASSIGN {=>{/* free(alien:Ident);*/
                                    FreeTokenStr(alien:Ident);
                                  };} .
> .
 rhs             = <
         = id:IDENT     { type := c_string;
                          Name :=id:Ident;
                          IValue := 0;
                          RValue:=0;}   .
         = nr:NUMBER    { type := c_int;
                          IValue := nr:Value; }  .
         = fl:FLOAT     { type := c_float;
                          RValue := fl:Value; }  .
 > .

IDENT            : [Ident       : pchar] { Ident       := NULL       ; } .
NUMBER          : [Value        : int ]   {  Value       := 0 ;} .
FLOAT           : [Value        : float ] {  Value       := 0 ;} .
ORDER            : [Value       : int ]   {  Value       := 0 ;}.
```

```
OPTION          : [Value      : int ]   {  Value      := 0 ;}.
RECUP           : [String     : pchar].


/*
MODULE tree
DECLARE
step
path            =             [tree: tTree SYN] .


END tree
*/
MODULE Params
DECLARE
rhs             = [type:int SYN] [IValue:int SYN] [RValue:float SYN] [Name:pchar
SYN]
                                        {type :=0; IValue :=0; RValue:=0; Name :=
NULL;} .
END Params
```

Figure A.1 – Context free grammar for Parser module

2. LEXICAL GRAMMAR OF TOKENS OF HTTP REQUEST

The following text represents the listing of lexical specifications of the Scanner module (see chap. 3 § 3.2.2) generated by the Rex processor **[6]**.

```
EXPORT {
# include ”Position.h”
#ifdef __cplusplus
  # include ”env_class.h”
#else
  # include ”env_var.h”
#endif
#include ”rMemory.h”
INSERT tScanAttribute
}
GLOBAL {
# define yyInitBufferSize 1048
# define yyInitFileStackSize 1

#include <string.h>

INSERT ErrorAttribute

#define BUFFLENGTH 50
static char OptBuffer[BUFFLENGTH] =” ”;




static unsigned char RecoverOption(char* InpStr)
{
if(strcmp(InpStr,”REFERER”)==0)  return optReferer;
if(strcmp(InpStr,”CONNECTION”)==0)      return optConnection;
if(strcmp(InpStr,”HOST”    )==0)        return optHost;
if(strcmp(InpStr,”FORWARDED”)==0)       return optForwarded;
if(strcmp(InpStr,”ACCEPT” )==0)         return optAccept;
if(strcmp(InpStr,”ACCEPT–LANGUAGE”)==0) return optAcceptLang;
if(strcmp(InpStr,”ACCEPT–CHARSET” )==0)  return optAcceptChar;
if(strcmp(InpStr,”ACCEPT–ENCODING”)==0) return optAcceptEnc;
if(strcmp(InpStr,”USER–AGENT”)==0)      return optUserAgent;
if(strcmp(InpStr,”UA–PIXELS” )==0)      return optUAPixels;
if(strcmp(InpStr,”UA–COLOR” )==0)return optUAColor;
if(strcmp(InpStr,”UA–OS”  )==0)         return optUAOS;
if(strcmp(InpStr,”UA–CPU” )==0)         return optUACPU;
if(strcmp(InpStr,”CONTENT–TYPE” )==0)   return optContType;
if(strcmp(InpStr,”CONTENT–LENGTH” )==0) return optContLen;
if(strcmp(InpStr,”AUTHORIZATION” )==0)  return optAuth;
if(strcmp(InpStr,”PRAGMA” )==0)  return optPragma;
if(strcmp(InpStr,”VIA” )==0)             return optVia;
if(strcmp(InpStr,”EXTENSION” )==0)      return optExtension;
return MAXENV –1;
}
#define yySetPosition


}
DEFAULT { if(EntitySize>0)EntitySize––;printf(”Scanner detected error in input,
illegal character code is : %d\n”,*(TokenPtr)); }
/*DEFAULT { MessageI (”illegal character”,xxWarning,Attribute.Position,xxCharacter,
                    TokenPtr); } */
DEFINE
   letter  = {A–Za–z} .
   digit   = {0–9} .
   id      = letter | (letter | digit | _ | \– |\+ | %)(letter | digit | _ | \– |\+ |
%) + .
   nr      = digit+  .
   lf      = \10 .
   cr      = \13  .
   sp      =\32 .
   slash    = / .
   point    = \. .
   sign    = (\–|\+) .
   commentchar = – {\10 \13 \0}.
    NULLCHAR = \0.

   hex    = ( digit | A|a|B|b|C|c|D|d|E|e|F|f) .
   flstr  = (sign )? nr point nr (E(sign)?nr)? .
```

```
START
   option optstring ending



RULES


#STD#  GET      :– {Attribute.ORDER.Value = httpGET; return(ORDER); }
#STD#  POST     :– {Attribute.ORDER.Value = httpPOST; return(ORDER); }
#STD#  slash      :– {return(SLASH);}
#STD#  point      :– {return(POINT);}
#STD#  HTTP     :– {return(HTTP);}
#STD#  sign ? nr  :– { Attribute.NUMBER.Value = ( long)atol (TokenPtr);
                        return (NUMBER); }

#STD#   flstr     :– { Attribute.FLOAT.Value = (float)atof (TokenPtr);
                        return (FLOAT); }
#STD#   id        :– { Attribute.IDENT.Ident = (char*)AllocTokenStr(TokenLength);
                      GetWord(Attribute.IDENT.Ident);
                         return (IDENT); }
#STD,optstring, option# NULLCHAR  :– {}
#STD# \&   :– {return(SEP);}
#STD# \=   :– {return(ASSIGN);}

#STD# \?   :– {return(URLSEP);}
#STD#\–           :– {return('–'); }
#STD#\+           :– {return('+'); }
#STD#\,           :– {return(','); }

#STD#    cr  lf   :– { yyStart (option);   }
#optstring#    cr  lf            :– { yyStart (option);    }
#optstring#    commentchar*      :–
{Attribute.RECUP.String=(char*)AllocTokenStr(TokenLength);
                              GetWord(Attribute.RECUP.String);
                              return (RECUP); }

#option#  id             :–      { GetUpper(OptBuffer);
                           Attribute.OPTION.Value = RecoverOption(OptBuffer);
                           return(OPTION); }
```

```
#option#  \:                     :–  { yyStart (optstring); }

#option#  cr  lf          :– { yyStart (ending);return(END_MAIN); }

#ending#  sign ? nr       :– { Attribute.NUMBER.Value = ( long)atol (TokenPtr);

                      /* TestTokensInEnding();*/

                       EntitySize = EntitySize–TokenLength;

                        return (NUMBER); }


#ending#   flstr  :– { Attribute.FLOAT.Value = (float)atof (TokenPtr);

                      /*  TestTokensInEnding(); */

                        EntitySize = EntitySize–TokenLength;

                         return (FLOAT); }

#ending#   id (point id)* :– { Attribute.IDENT.Ident =
(char*)AllocTokenStr(TokenLength);

                       GetWord(Attribute.IDENT.Ident);

                  /*     TestTokensInEnding(); */

                       EntitySize = EntitySize–TokenLength;

                         return (IDENT); }

#ending# \&       :– { /*TestTokensInEnding();*/

                EntitySize = EntitySize–TokenLength;return(SEP);}

#ending# \=       :– {/*TestTokensInEnding();*/

                  EntitySize = EntitySize–TokenLength;return(ASSIGN);}

#ending#  NULLCHAR       :– { if(EntitySize==0)return(EofToken); }

#ending#  cr  lf :– {/*printf(”AT THIS POINT THE ENTITY LENGTH IS:
%d\n”,EntitySize);*/

                  if(EntitySize==0)return(EofToken);}
```

Figure A.2 – Lexical specifications of Scanner module

Appendix B Configuration of Repository of Virtual file System

Server customisation consists in the design and generation of the Virtual File System of the server. This system is composed of three basic elements:

- tree–like data structure whose role is equivalent to the role of file system management tables. Through this data structure, called ***repository skeleton***, the user can find, read and modify the files embedded in the host environment,
- collection of routines which process this data structure,
- collection of memory regions storing the embedded files.

The repository skeleton is implemented as a web of interrelated records linked with the help of pointers. Its structure is described in the following context free grammar, specified in the language of the **ast** processor.

```
MODULE AbstractSyntax
/* this is a modified file – uniform processing of objects is implemented */

TREE data_base_struct

EXPORT  {

#include ”commondef.h”

typedef void(*tscript)(int,...);

 /* cleanup  of character chains created in the VFS */

/* types of file  nodes in VFS */

#define c_unknown 0
#define c_text 1
#define c_gif 2
#define c_jpeg 3
#define c_java 4
#define c_dir 5
#define c_script 6

}

GLOBAL  {
 #define yyALLOC(s1,s2) (tdata_base_struct)Alloc(s2)
```

```
             #define yyFREE(p,s) Free(s,(char*)p);

             #define closepchar(a) {if (a) close_char(a);}
            }
            /******* ABSTRTACT SYNTAX TREE DESCRIBING VIRTUAL FILE SYSTEM STRUCTURE ********/

            PROPERTY INPUT

            RULE

            repository =[name:pchar] <

               root   =  member_list:node  access_list [default_obj_name:pchar]
                         next:root .

               node   = next:node <
                             embedded_directory    =    member_list:node   access_list .
                             embedded_active_node  =    [file_id:ushort][nature:ushort]<
                               embedded_file              =    [content:pchar] [size:ushort]
              .
                               embedded_script     =    [script_exec:tscript] .
                             > .
               >.
            > .

             /* RESOURCE PATH ABSTRACT SYNTAX */

            path = [step:pchar] next:path .

             /* ACCESS LIST ABSTRACT SYNTAX */

            access_list = [user:pchar] [password:pchar] next:access_list .

            END AbstractSyntax
```

Figure B.1 – Structure of the repository skeleton

The ast processor **[9]** transforms this grammar into an abstract data type which is implemented in the module data_base_structure (see chap. 3 § 1.2). The routines used to generate and process this data structure are placed in the module data_base_processing (see chap. 3 § 1.3) and generated with the Puma processor of the Cocktail toolbox **[8]**. The routines from these two modules are used to build the VFS repository skeleton. An example of a skeleton constructing routine is presented in Figure B.1.

```
       tdata_base_struct db_page_repository(void)
       { tdata_base_struct lrepository;
         tdata_base_struct lptrstack[10];
          lrepository = InitRepository(NULL,”ROOT”,”page_root”);
          lptrstack[0]=BuildFileNode(”ROOT”,index_str,0,0,1);
          AppendNode(lrepository,lptrstack[0]);
          lptrstack[0]=BuildDirNode(”public”);
          lptrstack[1]=BuildFileNode(”gauge1”,Hello_page_str,0,1,1);
          InsertNode(lptrstack[0],lptrstack[1]);
          lptrstack[1]=BuildFileNode(”gauge2”,Hello_page_str,0,2,1);
          InsertNode(lptrstack[0],lptrstack[1]);
          lptrstack[1]=BuildFileNode(”di80_param_form”,di80_param_form_str,0,3,1);
          InsertNode(lptrstack[0],lptrstack[1]);
          lptrstack[1]=BuildFileNode(”dvc5000_1”,dvc5000_str,0,4,1);
          InsertNode(lptrstack[0],lptrstack[1]);
          lptrstack[1]=BuildFileNode(”dvc5000_2”,dvc5000i_str,0,5,1);
          InsertNode(lptrstack[0],lptrstack[1]);
          AppendNode(lrepository,lptrstack[0]);
          lptrstack[0]=BuildDirNode(”images”);
```

```
        lptrstack[1]=BuildFileNode(”alstom”,alstom_img,alstom_img_length,6,2);
        InsertNode(lptrstack[0],lptrstack[1]);
        lptrstack[1]=BuildFileNode(”DI80Mimic”,DI80Mimic_img,DI80Mimic_img_length,7,2);
        InsertNode(lptrstack[0],lptrstack[1]);
        lptrstack[1]=BuildFileNode(”ccd”,ccd_img,ccd_img_length,8,2);
        InsertNode(lptrstack[0],lptrstack[1]);
        lptrstack[1]=BuildFileNode(”HartMimic1”,HartMimic1_img,HartMimic1_img_length,9,3);
        InsertNode(lptrstack[0],lptrstack[1]);
        lptrstack[1]=BuildFileNode(”sensor”,sensor_img,sensor_img_length,10,2);
        InsertNode(lptrstack[0],lptrstack[1]);
        lptrstack[1]=BuildFileNode(”valve”,valve_img,valve_img_length,11,2);
        InsertNode(lptrstack[0],lptrstack[1]);
        AppendNode(lrepository,lptrstack[0]);
        lptrstack[0]=BuildDirNode(”javadir”);
        lptrstack[1]=BuildFileNode(”Trend”,Trend_bcode,Trend_bcode_length,12,4);
        InsertNode(lptrstack[0],lptrstack[1]);
        AppendNode(lrepository,lptrstack[0]);

         return lrepository;
     }
```

Figure B.2 – Example of skeleton constructing routine

This procedure builds the skeleton tree which spans the VFS repository presented as follows:

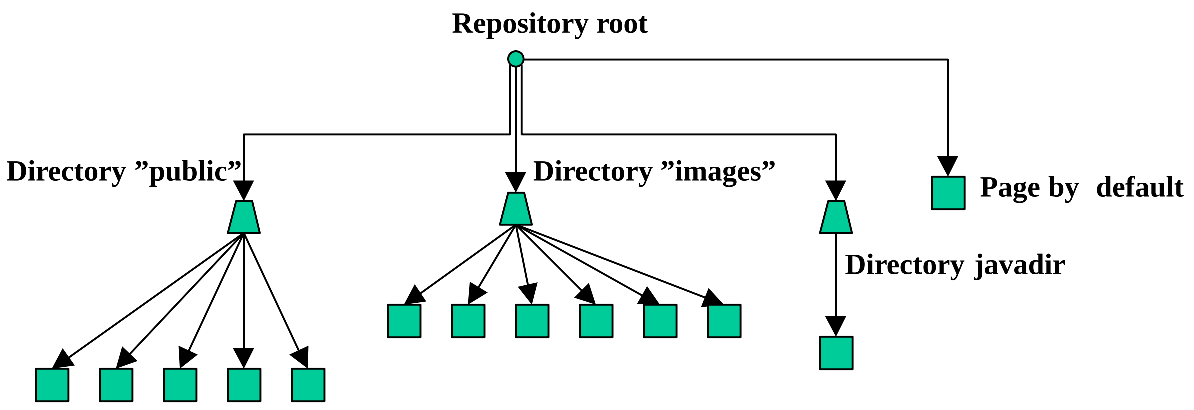


Figure B.3 – Virtual File Tree

The repository skeleton generated by the procedure contains:

- one by default page named **`ROOT`**:;
- directory **`public`** which contains five HTML pages: **`gauge1, gauge2, di80_param_form, dvc5000_1`** and **`dvc5000_2;`**
- directory **`images`** which contains six images in gif and jpeg format: **`alstom, DI80Mimic, ccd, HartMimic1, sensor`** and **`valve;`**
- director **`javadir`** which contains one file: **`Trend`**.

Appendix C — Embedding HTML Pages, Images and Applets

Embedded files containing passive server objects like HTML pages (or more precisely page templates), compressed images and bytecode of Java applets are referenced by the leaves of the skeleton tree which describes the server repository. These objects should be placed in well defined server memory areas and be efficiently accessed when the request for service is received by the server engine. Two methods of creation of such objects can be envisaged:

1. <u>Static</u>: the corresponding memory area is reserved and filled in at server build time ; the advantage of such a solution is its relative simplicity: the embedding process consists in transforming the objects into corresponding data structures expressed in C and in linking these objects with the server code; the disadvantage of such a solution is the rigidity of the server structure which cannot be extended at run time;
2. <u>Dynamic</u>: region reservation and filling in is done at run–time of the server program; the advantage of such a solution is its flexibility – the files can be installed on the running server; the price to pay for this feature is the increase in complexity of the server program which should support the means of loading files into memeory regions at run–time (e.g. via TFTP protocol).

The current version of the Field Device Web Server supports only the first method of file installation. The implementation of the second method can be envisaged without major problems, especially without the degradation of the currently supported features.

The embedded file can be transformed into a module which is linkable with the server main program. An example of such a file, embedding an image in ***gif*** format, is shown in the following figure:

```
extern const unsigned char aautobull2_img[];

extern int aautobull2_img_length;

const unsigned char aautobull2_img[] = {
0x47,0x49,0x46,0x38,0x39,0x61,0x0c,0x00,0x0c,0x00,0xb3,0xff,0x00,0xff,0xff,0x66
 ,0xff,0xff,0x33,0xff,0xff,0x00,0xcc,0xff,0x00,0xc0,0xc0,0xc0,0x99,0xff,0x00,0x99
 ,0xcc,0x00,0x99,0x99,0x00,0x99,0x66,0x00,0x66,0x99,0x00,0x66,0x66,0x00,0x33,0x66
 ,0x00,0x33,0x33,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x21,0xf9,0x04
 ,0x01,0x00,0x00,0x04,0x00,0x2c,0x00,0x00,0x00,0x00,0x0c,0x00,0x0c,0x00,0x00,0x04
 ,0x42,0x90,0xc8,0x49,0x6b,0xbb,0xb7,0x92,0x86,0x42,0x10,0x47,0x43,0x35,0x02,0xe0
 ,0x09,0x83,0x21,0x6e,0x88,0x79,0x0e,0x83,0x22,0x92,0x81,0x39,0x08,0xc6,0x91,0xcc
 ,0xde,0x57,0x14,0xba,0xdd,0x46,0x40,0x84,0x19,0x14,0x0b,0xd9,0xe6,0x00,0x1b,0x1c
 ,0x16,0x50,0xc6,0xaa,0x31,0x28,0x18,0x12,0x48,0xe9,0x48,0xa1,0x53,0x68,0x2d,0x98
 ,0x95,0x24,0x02,0x00,0x3b};

int aautobull2_img_length = 149;
```

/* ● */

Figure C.1 – Module representing the code of an embedded image (shown within the comment)

Appendix D Serving dynamic HTML

The packages of the Field Device Web Server provide two methods to generate pages whose context depends on the data retrieved from the basic application called in the sequel ***dynamic pages***. The first of them concerns the pages which have only a few variable elements placed within a large body of invariant data. The second concerns genuinely dynamic pages containing many variable components the parameters of which vary frequently in time.

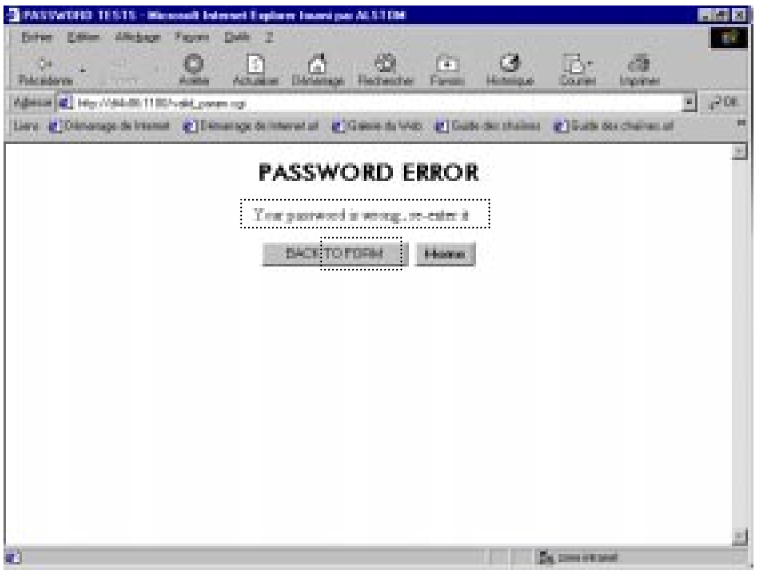
1. SKELETON BASED DYNAMIC PAGES

The first type of page should be structured into two components:

- page skeleton – page invariant frame which can easily be edited using standard tools like MS Front Page ™ ; this part of the page can represent either the informative part of the page or the basic components of the interactive part of the page;
- variable components – page elements which change page aspect (font colours, parts of text, references to images, type of interactive controls, applet parameters, etc.) or can change browser behaviour while processing the page (parameters of page header, applet parameters, hyperlink contents, etc.); variable components are represented by format expressions in C language (%s for strings, %d for integers, etc.).

The process of placing variable components within the page skeleton can be implemented either by the direct modification of HTML code of the skeleton or in some cases can be handled during edition of the skeleton on the specialised tool. The following example shows the piece of HTML code, which represents the page skeleton with an inserted variable component.

The HTML page must be generated so as to resemble the page in Figure D.1 with the possibility of varying three elements:


PASSWORD ERROR

Your password is wrong, re-enter it

Figure D.1– View of the page from the example

1. Error message ***1*** shown by the upper frame;
2. Part of text ***2*** on the left button shown by the lower frame;
3. Operation corresponding to a click on the left button (invisible in the figure).

The analysis of the problem shows that it is enough to provide the page template with three variable elements, all three of string type. The page template will be laid out as in Figure 3.6. Three formats corresponding to three variable parameters are shown in Figure D.2 as white fields.

```
<html>

<head>
<title>PASSWORD TESTS </title>
</head>

<body>
<script Language=”JavaScript”><!--
function backhome_onclick() {
location=”%s”
}
//--></script>
<form method=”POST” action=”/public/di80_param_form.htm”>
  <div align=”center”><center><p><strong><font face=”FuturaA Md BT” size=”5”>PASSWORD
ERROR</font></strong></p>
  </center></div>
  <p align=”center”> %s &nbsp;&nbsp;&nbsp; </p>

<p align=”center”><input type=”submit” value=”BACK %s” name=”SendButton”>
&nbsp;<INPUT    id=backhome    language=javascript    name=backhome    onclick=”return
backhome_onclick()”
 type=button value=Home style=”font-family: FuturaA Md BT; font-weight: bold”></p>
</form>
</body>
</html>
```

Figure D.2 – Skeleton of the page from the example

The section of program which makes this page resemble Figure D.1 is as follows:

```
        ……………………..
        char  strlock[10] ;
        char strerrmess[40];
        char* strbuttext[10];
        ……………………..
        strcpy(strolck, ”/”);
        strcpy(strerrmess,  ”Your  password  is
        wrong, re–enter it”);
        strcpy(strbuttext, ”TO FORM”);
        ……………………..
        status    =    sockprintf    (sock_id,
        page_template,
        strlock,strerrmess,strbuttext);
        ……………………..
```

Figure D.3 – Part of Figure D.1 program

This section of the program makes use of the routine **`sockprintf`** from the module ***sockinterf***. (see chap. 3 § 2.3.5.). It is also assumed that the variable **`page_template`** takes on the value of the character string representing the page template from Figure 3.6.

2. ON–LINE GENERATION OF PAGES

Pages of the second category are generated by the programs which make use of routines from package ***HTML GENERATION*** (see chap. 3 § 4.). These routines generate the structures of data which are the abstract representations of the HTML pages and convert them to character strings inserted into the server's response PDUs. The method gives more flexibility than the one previously described, since every parameter can be modified from one procedure call to another. Figure D.4 shows an example of a procedure which constructs the abstract representation of the page shown in Figure 3.5.

```
#include ”html_gen_hl.h”
#include <malloc.h>
#include <string.h>
thtml_page_elements GenHtmlPage_PASSWORD_TESTS_0(char* lstrlock, char* lstrmess,
                                                  char* lstrbut)
{
 thtml_page_elements lpage;
 thtml_page_elements lptrstack[10];
 char* lbuff = malloc(100);
 sprintf(lbuff, (”\nfunction backhome_onclick() {location=\”%s\”}”, lstrlock);

   lpage = BuildHtmlPage(”PASSWORD TESTS ”,NULL,NULL,0,NULL,NULL);
   lptrstack[1] =_T(lbuff);
   lptrstack[0] =create_script(“JavaScript”,1,”lptrstack[1]);
   lpage =       AppendObjectToPage(lpage,lptrstack[0]);
   lptrstack[5] =create_formatted_text(NULL,5,0,0,0,1,_T(”PASSWORD ERROR”));
   lptrstack[4] =create_formatted_text(NULL,0,1,0,0,1,lptrstack[5]);
   lptrstack[3] =create_paragraph(NULL,1,lptrstack[4]);
   lptrstack[2] =create_paragraph(”center”,1,lptrstack[3]);
   lptrstack[1] =create_paragraph(NULL,1,lptrstack[2]);
   lptrstack[2] =create_paragraph(”center”,1,_T(lstrmess));
   sprintf(lbuff, “BACK %s” lstrbut);
   lptrstack[4] =BuildValuedInput(28,”SendButton”,lbuff);
   lptrstack[5] =BuildButtonInput(”backhome”,”Home”,”return backhome_onclick()”);
       lptrstack[3] =create_paragraph(”center”,2,lptrstack[4],lptrstack[5]);
       lptrstack[0]                                                         =
     make_html_form(”POST”,”/public/di80_param_form.htm”,3,lptrstack[1],lptrstack[2],
     lptrstack[3]);
       lpage =       AppendObjectToPage(lpage,lptrstack[0]);
       free(lbuff);
       return lpage;
     }
```

Figure D.4 – Example of abstract representation procedure

The data structure produced by this routine can be converted to the character string by the call to the procedure `HtmlPageToString` (see chap. 3 § 4.4.1.) as in the piece of code below:

```
 ……………………………….
char* template_string;
   ushort template_length;
   thtml_page_elements thepage;


    template_string = malloc(4096);
    thepage = GenHtmlPage_PASSWORD_TESTS_0 (“/”,”Your password is wrong, re–enter it,
                                           ”TO FORM”);
    template_length =HtmlPageToString(template_string,0,
                            thepage);
……………………………….
```

Figure D.5 – Example of HtmlPageToString routine

Glossary

Applet	Self standing Java programs whose reference is included in HTML pages; applets are downloaded from the server and executed within the browser.
Authentification procedure	Distributed program which implements the access control to server resources based on the exchange the via network of user name accompanied by its secret password. There are different types of authentification procedures; the simplest one, called basic authentification, consists in transmission of a pair of data (user name + user password) coded by a widely known 64–base coding algorithm. The basic authentification offers only rudimentary access control and is a major security loophole in the protection of INTERNET sites.
Base64 encoding	Coding algorithm used in the basic authentification procedure to encrypt username and password sent in HTTP Request PDU; according to this algorithm the characters of the original string are by the handled elements of the standardised encoding vector.
Basic application	The application which fulfils the principal mission of Alspa 8000 equipment; e.g.: FIP/HART protocol conversion is the basic application of DI80.
BSD sockets	Berkeley Software Distribution sockets – software component implementing the standard presentation interface to different transport protocols; sockets are mostly used on top of the TCP/IP stacks.
Cocktail	Compiler Compiler Toolbox Karlsruhe – a collection of co–operating highly efficient software tools used to generate compiler modules.
Embedded image	Image coded in one of the well known compact formats (GIF, JPEG) and stored within the embedded file system of host equipment.
Embedded server	Servers designed to be placed in small devices; their characteristics are: small code size, no disk storage and no sophisticated software interfaces such as interface to data base or interface to script languages.
HTML	HyperText Markup Language – a variant of SGML, a meta–language used to describe the layout of printable documents.
HTTP	Hypertext Transfer Protocol – one of Internet’s most used application protocols. It is based on TCP/IP and is used to transport hypertext documents.
Internet	Interconnected Networks – a world wide communication infrastructure composed of many thousands of interconnected Local Area Networks. The networks share the same communication technology based on the pair of protocols: **TCP** and **IP** (*Transmission Control Protocol* and *Internet Protocol*). INTERNET technology has its origins in a DOD supported project ARPA. The results of this project were made public in the 1980s and were followed by the ever growing arrival on the market of open network applications.

Internet browser	Internet client agent implementing the principle of exploration of hypertext documents, are *document browsing*, by following the hyperlinks that included in the documents.
LR(1)	Class of abstract grammars easy to parse by efficient analysers.
Parser	Program which analyses a phrase of a text and states whether the structure of this phrase conforms to the rules of an abstract grammar describing the language of the phrase; parsers are essential parts of language compilers; parsers are usually hand generated automatically. In this project the parser verifies the syntax of incoming HTTP request PDUs and co–operates with programs which compose the response PDUs.
PDU	Exchanging between communicating parties during service implementation; in the Internet communication model there are exist two classes of PDUs: *request PDUs* sent by the client and *response PDUs* furnished by the server.
Portable TCP/IP stack	Set of co–operating software components which implement the specifications of TCP and IP protocols in a way making them independent of the underlying data link; the version of the portable stack used in this project can be used indifferently on WorldFIP and on Ethernet links; the stack includes also the routing function.
Scanner	Program which prepares the efficient operation of the Parser by organising the input stream of characters into aggregates called tokens; scanner are rarely hand written; in the majority of cases they are generated by specialised tools like ***lex*** from standard Unix delivery or ***rex*** from Cocktail toolbox.
Token	Aggregate of characters representing a lexical atom of a formal language; well known examples of tokens are keywords, indentifiers, numbers, separators, comments, etc. The notion of token simplifies and clarifies the description of formal languages by means of context–free grammars; tokens are generated from the input character streams by scanner programs.
Virtual File System	Internal data structure of an embedded server implementation installed on a diskless hardware platform, which lets the server application imitate the file system of a device with the mass storage; this structure is sometimes referred to as ***VFS*** or ***embedded file system***.
WWW	(World Wide) Web – the aggregate collection of hypertext documents retrievable via Internet by means of the HTTP protocol.