iMatix Corporation

iMatix Studio

an iMatix Technical White Paper

21 February, 1998

iMatix Studio Technical White Paper

Disclaimer

The information contained in this document is distributed on an "as-is" basis without any warranty either expressed or implied. The customer is responsible for the use of this information and/or implementation of any techniques mentioned. iMatix has reviewed the information for accuracy, but there is no guarantee that a customer using the information of techniques will obtain the same or similar results in its own operating environment.

It is possible that this material may contain references to, or information about, iMatix products or services that have not been announced. Such references or information must not be construed to mean that iMatix intends to announce such products or services.

iMatix retains the title to the copyright in this paper, as well as title to the copyright in all underlying works. iMatix retains the right to make derivative works and to republish and distribute this paper to whoever is chooses to.

Trademarks

iMatix, **iMatix Studio**, **Libero**, **Xitami**, **SMT** and **SFL** are trademarks or registered trademarks of iMatix Corporation. **ETK** is a trademark of TOSC, Intl. Other trademarks are the property of their respective owners.

Authored by

iMatix Corporation
e-mail: info@imatix.com website: http://www.imatix.com

European Headquarters
9 Pijlstraat, 2060 Antwerpen, Belgium
Tel. +32-3-231-5277, fax. +32-3-231-9877

Version Information

Document version 1.c.
Document reference: twp9801.
Release date: 21 February, 1998.

Contents

1. Introduction

1.1 About iMatix Corporation

iMatix Corporation is a privately-held US-registered corporation with operations in the US, Europe, Australia, and New Zealand. Our headquarters are in Belgium; we have users in more than 50 countries.

Since 1995, iMatix has been working on the research and development of an Internet-based approach to software development. We believe that Internet technology offers the only viable long-term approach to the complex and expensive problem of developing large-scale business applications.

While investing heavily in leveraging the many Internet technologies for our own ends, we have also been developing a business model based on those same technologies. iMatix was distributing free software before this became a fashionable way to create new markets. Our products, technologies, and reputation reach many thousands of key technical people through popular download sites.

1.1.1 Our Experience With Business Software Development

Our experience in the construction and use of large-scale software development tools dates from 1985. Between 1985 and 1997, we worked on software development tools (ETK) that were used to construct large applications in Europe and the US: tour operator systems, accounting packages, financial management packages, and many other core business applications serving hundreds or thousands of users. These applications shared some interesting and valuable aspects:

- They were portable between an astonishing range of systems, from Digital VMS to Data General MV, IBM S/38, IBM AS/400, IBM MVS, Unix, Windows NT, and even MS-DOS in some cases.

- They have proven to be maintainable and extensible far longer than was planned; most will be used into the next century without difficulty. In a typical instance of the foresight with which we built these tools (and which was rare in 1985), dates were always stored with the century, so these applications are year-2000 compatible by default.

In building these tools, and supporting client applications on a variety of operating systems, we learnt some important (and perhaps obvious) lessons:

1. Portability, especially to unknown future systems, is the key to application longevity. The rapid rate of technological change means that any platform-dependent approach to software development is inherently unstable, due to the inevitable demise of any given platform.

2. Control over, and mastery of, the technical environment is essential when it comes to answering new and unforeseen needs, such as moving the application to a new database or making connections to new external systems.

3. Application developers need a well developed environment so that they can focus on the purely functional aspects, and not waste time solving technical problems. Most developers are not able to provide good solutions to technical problems. A separation of these two concerns is therefore very important.

4. An unstable, or untested technical platform will cause serious, sometimes fatal delays for an application development. The benefits of using new, untested technology are always outweighed by the risks, unless these risks are managed from the start. One way of doing this is to encapsulate the new technology so that developers are neither exposed to it, nor tied to it.

5. Applications that serve many users require support from a transaction processing system so that operating system resources are efficiently used.

The requirements of the end-user in terms of interface and connectivity have also evolved considerably, from local plain-text terminal interfaces, to richer GUI interfaces across wide-area networks, while the actual functional needs have not changed that significantly.

1.2 What Is iMatix Studio?

Today, the de-facto standard user interface - the web browser - is richer than a plain text terminal application, but simpler than a GUI word processor. It can show moving images and rich information layouts, yet works within a single window, with no pop-ups, and no complex operations beyond point and click. The extraordinary simplicity and availability of the web browser makes it more than just a way to browse interesting web sites. We are not the first to note that the web browser has retaken the role of the mainframe or minicomputer terminal, albeit with a much improved visual and connectivity metaphor.

iMatix Studio combines a decade of experience in large-scale software development approach with the technology of the Internet and the web browser to provide a technical platform for stable, reliable, and economic web-based business applications.

The purposes of this white paper are to:

- State the design objectives of iMatix Studio.
- State the current alternatives to a tool like iMatix Studio.
- Describe the iMatix Studio architecture.
- Present how iMatix Studio can give a complete solution to the problem of designing and developing solid software applications.

2. Design Goals and Requirements

Before we note our design goals and requirements, we can examine our representative business application:

1. The application serves several hundred or thousands of internal or external clients.

2. The application runs on a central server system, typically running Unix, OpenVMS, Windows NT, or OS/2, possibly based on a cluster of servers.

3. The application works across a local area network, but across a wide-area network to reach users in other regions or countries.

4. The application is based on a large, complex, and critical database, usually built on a relational database such as Oracle, DB2, Ingres, etc.

5. The application is developed over a period of years, perhaps decades, and represents a major investment for the company.

The following requirements were designed to address the various challenges of developing such applications:

- **Client, server, and database portability** - The developed applications should be deployable on any mixture of client or server system, and using any suitable database at little or no extra cost.

- **Reliability and quality** - The developers should be given a framework, or environment, in which it is self-evident to develop reliable and efficient programs. The provision of a solid, good, framework is essential to developer confidence and productivity.

- **Economy** - The developed applications should be able to run on modest client and server systems, across low-speed networks and on easily-available databases. For instance, the choice of a browser-based interface is made mainly because it is extremely cheap in terms of development, deployment, training, and support.

- **Scalability** - The developed applications must also be able to handle many thousands of users, on the largest of systems, without inherent bottlenecks. Even if this is not the intention when an application is designed, it is a truism that a successful application always risks being killed by its inability to handle its own success.

- **An open, controllable design** - The technical environment must be open, accessible, and controllable. For example, the specific implementation on any platform must be within the control of the application designer, so that design decisions can be enforced at the lowest level.

- **Simplicity** - The technical environment should be simple enough to learn within a few days and fully master within a few weeks at most.

3. Design Alternatives

3.1 Analysis Of Client-Server Approaches

We stated that the technical basis for our design should be the technology of the Internet. This section presents a detailed discussion about the alternatives, and the reasons why we made this fundamental choice.

3.1.1 The Client-Server Landscape

Figure 1 shows one way of looking at the client-server landscape. All software applications running on a mix of client and server systems – from pure legacy applications to stand-alone PC programs – can be mapped onto this figure:

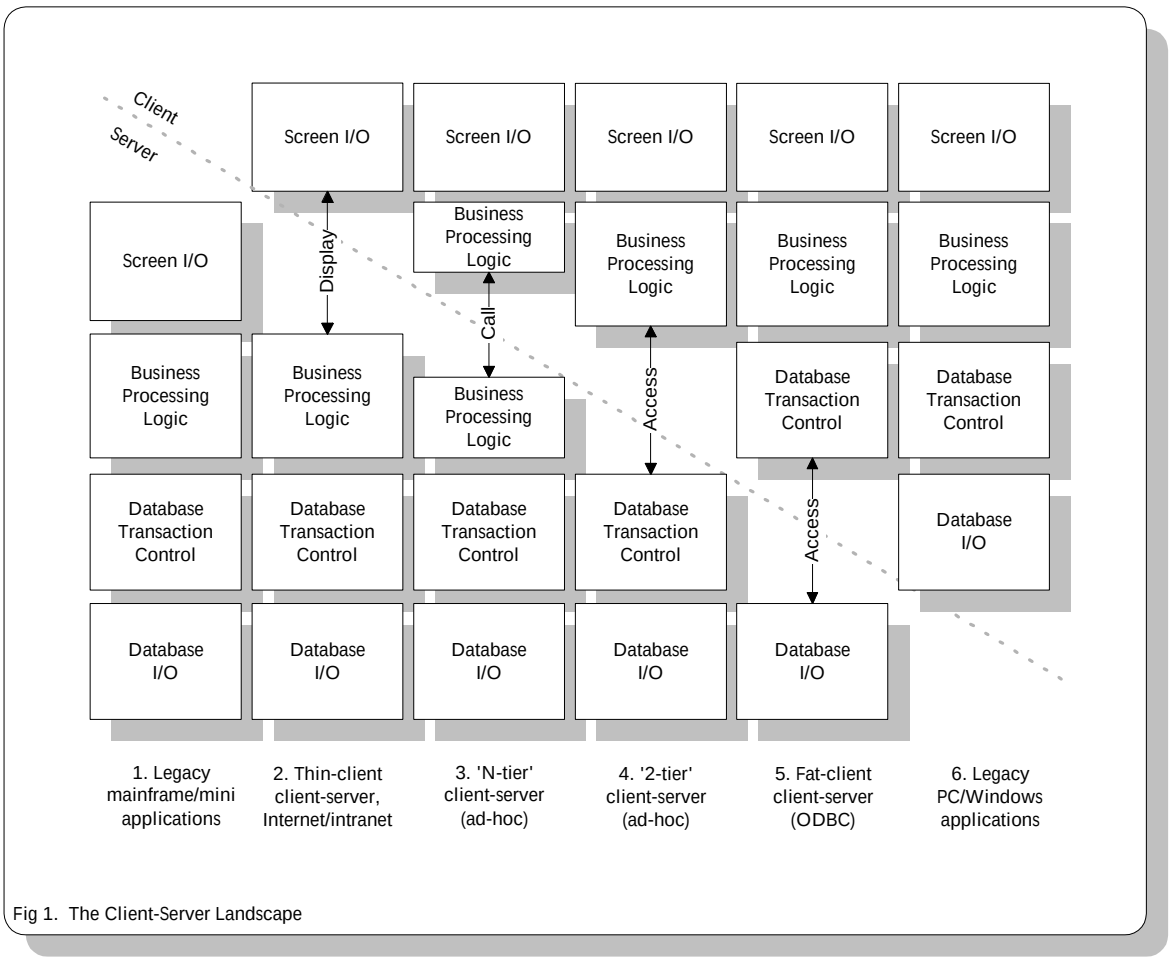


Fig 1. The Client-Server Landscape

In our analysis, we considered these general guidelines to cost and efficiency:

- *Server-side code is cheaper than client-side code.* Our experience has taught us that server-side code can be (there are no guarantees) more stable, more portable, and of a

higher quality than client side code. Those with a PC background (e.g. Microsoft) may argue the opposite. We note simply that it is possible to provide a very clean and encapsulated environment for large-scale application server code while this is much harder for PC code, which is usually developed as single-person projects.

- *It is expensive to distribute software*. When application components run on client systems, they must be installed, and managed. Even when the issue of distribution is solved, other issues remain. For example, it is very difficult to allow a user to run two different versions (e.g. test and production) of an application. Often, the client system (e.g. a PC) must correspond to a certain configuration (e.g. Windows NT client, Pentium, 32 Mb,…). This problem is worst when proprietary development tools are used as they may even impose specific versions (NT 4.0 SP3 but not NT 5.0). Such constraints can turn into serious problems when multiple applications are used on the same PC (one requires NT 5.0, and one requires 4.0SP3…).

- *Network traffic affects response time*. An application that sends large amounts of data across a network will inevitably run slowly; this problem scales as the number of users, or their distance from the server, increases.

- *Database traffic is an order of magnitude larger than user-interface traffic.* In other words, what is shown on the screen is often the synopsis of a much larger amount of data handled by the program. To reduce network traffic (and therefore improve response time), the database traffic should not pass across the network. We call this '*database proximity*'.

- *Generic screen I/O is cheap.* A major hidden cost of GUI development is the effort required to build each specific application screen. Even with modern GUI tools, this can consume much more time than the actual functional programming. A generic screen I/O layer simplifies the user-interface, and provides a high-level abstraction that makes screen design a fast process driven by the functional needs, not cosmetic considerations.

- *Complexity is expensive; simplicity is cheap.* People are not generally good at handling complexity.

- *Text-only UI's do not sell; people need a GUI.* Beside all other considerations ('but do we *really* need a spinning logo?'), a GUI can show more information, more usefully, than a text UI.

3.1.2 Comparing The Different Approaches

We considered each of these criteria, five stars indicates excellent, while one star indicates poor performance or high cost:

	1. Legacy Mainframe	2. Thin Client	3. N-Tier C/S	4. 2-Tier C/S	5. Fat Client	6. Legacy PC
Server-side code	*****	*****	***	***	*	*
Distribution	*****	*****	*	*	**	***
Network load	*****	*****	***	***	*	*****
Database proximity	*****	*****	*****	*****	*	*****
Generic screen I/O	*****	*****	*	*	*	*
Simplicity	*****	*****	*	*	***	***
GUI	*	****	*****	*****	*****	*****

Comments:

- Approach 1 (legacy mainframe) does not use the client system except as a dumb terminal or terminal emulator.

- Approach 2 (thin client) requires communications between the screen I/O layer and the business processing code. If this must be handled by each programmer independently, clearly it will be a very expensive method. However, we know that it is possible to *abstract* the screen I/O and communications issues so that the programmer has no extra work at all.

- The network overhead is highest in approaches 3, 4, and 5. These work by sending SQL commands to the server and receiving replies. In simple cases this works fine. But for complex displays, many dozens or hundreds of records must be read to display a small amount of data. Take the basic example of an application program that searches a set of records for the best match. Approaches 1 and 2 work much more efficiently on networks. Approach 6 does not really need a network at all, except for software distribution.

Only approach 2 (thin client) allows full abstraction of the screen interface, and consequent savings in both network overhead, development cost, and overall system cost and stability. The rapid success of the Internet World Wide Web (which is good example of approach 2) proves that this is cheap, practical, and effective.

3.2 Alternatives Considered

There are many ways to write Internet and intranet client-server applications. The obvious question is: why did iMatix Corporation develop iMatix Studio, when so many tools are already available on the market?

The simple answer is that nothing existed that could do what we wanted at a reasonable price. Given our analysis that the web-based thin-client model offered the best value for money over the long term, we looked for tools that provided this model. What we found was that:

1. Pure Internet tools are mostly Java-based, a technology that is unstable and unproven.

2. The few proven and stable tools are either very costly, or are simply not designed for Internet applications.

3. Those tools that are reasonably priced and can deliver web-based applications are not designed for large-scale work.

4. Too many approaches depend on rapidly changing and unreliable technology, and proprietary products and protocols.

In the next section, we examine some alternatives for the development of a typical intranet database application, and explain our analysis of each alternative.

3.2.1 The Microsoft Alternative

The "Microsoft Alternative" is to write the application as ActiveX components, using Visual Basic, Visual C++, Visual J++, or JavaScript, and MS SQL Server and MS Transaction Server. The server systems would be Windows NT and the client systems would be Windows NT (4.0 or 5.x) or Windows 95. The following analysis is based on long and extensive experience with Microsoft products.

Disadvantages:

- Requires highly-skilled developers, familiar with a complex and rapidly-changing technical environment.

- Generally a very closed environment, with little recourse to third-party products except those tolerated by Microsoft.

- Very dependent on versions = unstable = expensive.

- Not a high-quality solution; network traffic is high, and database transactions are done in a naïve manner.

- No guarantee that the application will survive the rapid rate of technological change.

Advantages:

- High-level of developer control (writing in C or C++).

- 'Politically correct'; Microsoft is the IBM of the 1990's.

3.2.2 The Oracle Alternative

The "Oracle Alternative" is to write the application using Oracle Designer, and the Oracle database. The server system could be any supported UNIX or Windows NT system, and the client systems would be Windows NT or Windows 95. An Oracle Designer application can be converted into an intranet application that is implemented as a set of Java classes. The following analysis is based on experience with Oracle products including Oracle Developer and Oracle Designer in several projects.

Disadvantages:

- Requires fairly skilled developers.
- Imposes a proprietary database and programming language.
- Quite dependent on specific versions = expensive[1].
- A closed solution: for instance, it is a complex, slow, and delicate process to interface a Designer application to code written in other languages.
- The intranet Java solution is extremely slow, unless the Java classes are pre-installed on each client system. The model is excessively slow on wide-area networks.
- Configuration management is expensive in a wide-area network.

Advantages:

- Oracle tools are good for database-type applications.
- Oracle Designer provides a rich development environment with fair support for code reusability.

3.2.3 The PowerBuilder Alternative

The "PowerBuilder Alternative" is to write the application using PowerBuilder and a supported database. The server system could be any supported UNIX or NT system, and the client systems would be Windows NT or Windows 95 systems. The following analysis is based on experience of projects written using PowerBuilder at client sites.

Disadvantages:

- Requires highly-skilled developers.
- Does not scale for large systems: transaction management is weak.
- Imposes a proprietary database, programming language.

---

[1] For instance, on a recent project where the author participated, the client had to use 2 NT PCs, one running NT 3.51 for one application, and one running 4.0 for another application.

- Very dependent on versions = unstable = expensive.
- Configuration management is expensive in a wide-area network.
- Not designed for intranets.

Advantages:

- Gives a high level of control over the user-interface.

3.2.4 The Netscape Alternative

The "Netscape Alternative" is to write the application using the Netscape NSAPI protocol, in C. The server system could be any supported UNIX system and the client systems would be any Web-enabled system. The following analysis is based on our understanding of Netscape products, as well as some in-depth experience with the protocols and languages involved.

Disadvantages:

- Requires highly-skilled developers, with knowledge of NSAPI, HTTP, HTML, CGI, and JavaScript, and familiar with a complex and rapidly-changing technical environment.
- Does not scale for large systems: provides no transaction management, and is not multithreaded.
- Does not include any notion of toolkit; developers must create every tool from scratch.

Advantages:

- Free, or available at low cost.
- Really designed for the Internet.
- Support for programs written in almost any language (Java, C, C++, Perl, Python,…).
- Open: can be extended and enhanced in various ways.

3.2.5 The Lotus Notes Alternative

The "Lotus Notes" alternative is to write the application using Lotus Notes. The server would be an Windows NT, OS/2, Unix, or AS/400 and the client would be Windows 95, NT or OS/2, or any platform running a web browser. The following analysis is based on information gathered from a two-day course in Lotus Notes. It is likely that experienced Lotus Notes developers would have other opinions.

Disadvantages:

- Requires skilled developers.
- An expensive solution.

- Not designed as a high-volume application server.
- Not intended for application development; rather, intended as document-management, *groupware* platform.
- Proprietary.

Advantages:

- Suitable for some types of intranet applications, e.g. document management.
- Portable; can be deployed on the majority of systems.
- Works with HTML browsers as well as the specific Notes client.
- Can be programmed in LotusScript, Java, C, C++.

3.2.6 The Tuxedo Alternative

The "Tuxedo Alternative" is to use the Novell Tuxedo transaction manager as a technical platform. The server would be any suitable UNIX platform, while the client could be anything from a simple terminal to a Windows NT PC. The following analysis is based on second-hand knowledge of Tuxedo, and extensive experience with similar products (e.g. Digital ACMS, IBM CICS, Bull TDS).

Disadvantages:

- Not an Internet/intranet solution; there is no way to use Web technology in such a design except by building your own components.
- Does not provide any type of software development tool.
- Requires skilled developers.
- Expensive.

Advantages:

- Well-tested and robust technology.
- Scaleable; suitable for large applications.

3.2.7 The Java Alternative

The "Java Alternative" is to write the application as a set of client-side Java applets, or as distributed Java objects (also humorously called JavaBeans). The following analysis is based on knowledge about Java collected from technical journals, as well as experience in using and deploying the language.

Disadvantages:

- Requires highly-skilled developers.

- Produces slow applications.
- Highly unstable technical platform.
- Subject to an ongoing trade war between Sun, Netscape, and Oracle, and Microsoft.

Advantages:

- Freely or cheaply available.
- Simpler to learn than C++.

3.2.8 Comparing The Alternatives

The following tables compare the various alternatives (we add iMatix Studio for comparison):

	Proprietary	Tools	HTML?	C/S Model	DB
Microsoft	Yes	Basic	Yes	N-tier	SQL Server
Oracle	Yes	Yes	No	2-tier	Oracle
PowerBuilder	Yes	Basic	No	N-tier	Informix
Netscape	No	No	Yes	N-tier	Any
Lotus Notes	Yes	Basic	No	N-tier	Lotus Notes
Tuxedo	No	Basic	No	Mainframe	Any
Java	Yes	Yes	No	2-tier	Any
iMatix Studio	*No*[2]	*Yes*	*Yes*	*Thin client*	*Any*

	Portable	To?	Complexity	Languages
Microsoft	No	-	High	VB5.0 Visual C++ Java++
Oracle	Yes	Unix, NT	Medium	PL/SQL
PowerBuilder	No	-	High	PowerBasic C++
Netscape	Yes	Unix, NT	High	C, HTML, Java, Perl
Lotus Notes	Yes	-	Medium	LotusScript, C, C++, Java
Tuxedo	Yes	Unix	Medium	C, Cobol
Java	Yes	Many	High	Many
iMatix Studio	*Yes*	*Many*	*Low*	*C, Cobol, C++, Java, Perl*

[2] iMatix Studio is based on a set of open technologies, such as HTTP, SFL, Libero, WTP. The tools themselves are proprietary.

Note that these tables are based on information taken from various sources, and may be subjective or incorrect. This is an area that changes rapidly and where points-of-view are often partisan. We have tried to be as objective as possible.

This is how we judge each solution on our basic design criteria (iMatix Studio scores high marks by definition):

	Portability	Reliability	Economy	Scalability	Openness	Simplicity
Microsoft	*	**	**	**	**	**
Oracle	***	***	***	***	*	***
PowerBuilder	*	**	*	**	**	**
NetScape	***	***	*	***	****	*
Lotus Notes	**	****	***	**	*	***
Tuxedo	****	****	****	*****	****	***
Java	***	**	***	***	***	**
iMatix Studio	*****	*****	*****	*****	*****	*****

3.2.9 Conclusions

These analysis are not meant to be definitive; rather, we have highlighted the best and worst points of the various tools we have encountered during many projects. It is evident that the worst tool in the hands of a skilled practitioner will produce better results than the best tool in the hands of a fool.

However, the technical layers chosen to support an application development *must* help, not hinder the process.

We drew these conclusions from our study of available tools:

- The only alternatives that allow reuse of Web technology are Microsoft, Netscape.
- The only alternative that provides a high-level programming development process is Oracle. The other alternatives assume that each developer will program from scratch, from a blank page, or a simple template.
- The only alternatives that are really open are Netscape, Tuxedo, and Microsoft to some extent. The other alternatives oblige a proprietary, closed solution.
- The only alternatives that permit a thin-client approach are Lotus Notes, Netscape, and Tuxedo. The other alternatives require expensive, specific PC platforms (at least Windows 95, and generally Windows NT), and provoke high configuration management costs.

- Only Oracle provides any kind of formal CASE tool approach with a dictionary and code generators. Our experience shows that CASE tools and code generators can be a major factor in reducing development cost, errors, and maintenance cost.

- Almost all available approaches are far too complex. Complexity is generally a sign of a weak design, and this is most evident in approaches that are built out of a mixture of unrelated technologies that just happened to be available. While complexity is often marketed as 'features' or 'power', we believe it is better translated as 'high learning curve', 'expensive maintenance'.

So, our final design combined the best of these alternatives. Namely, a clean solution designed for the intranet from the start, based on solid tools to generate code, using well-established techniques to provide portability, and scalability for industrial-scale applications.

4. Overview of iMatix Studio Architecture

This chapter provides an overview of the iMatix Studio architecture. It is moderately technical; we assume that the reader has experience with building or using information systems.

This figure shows the general operating principles of iMatix Studio:

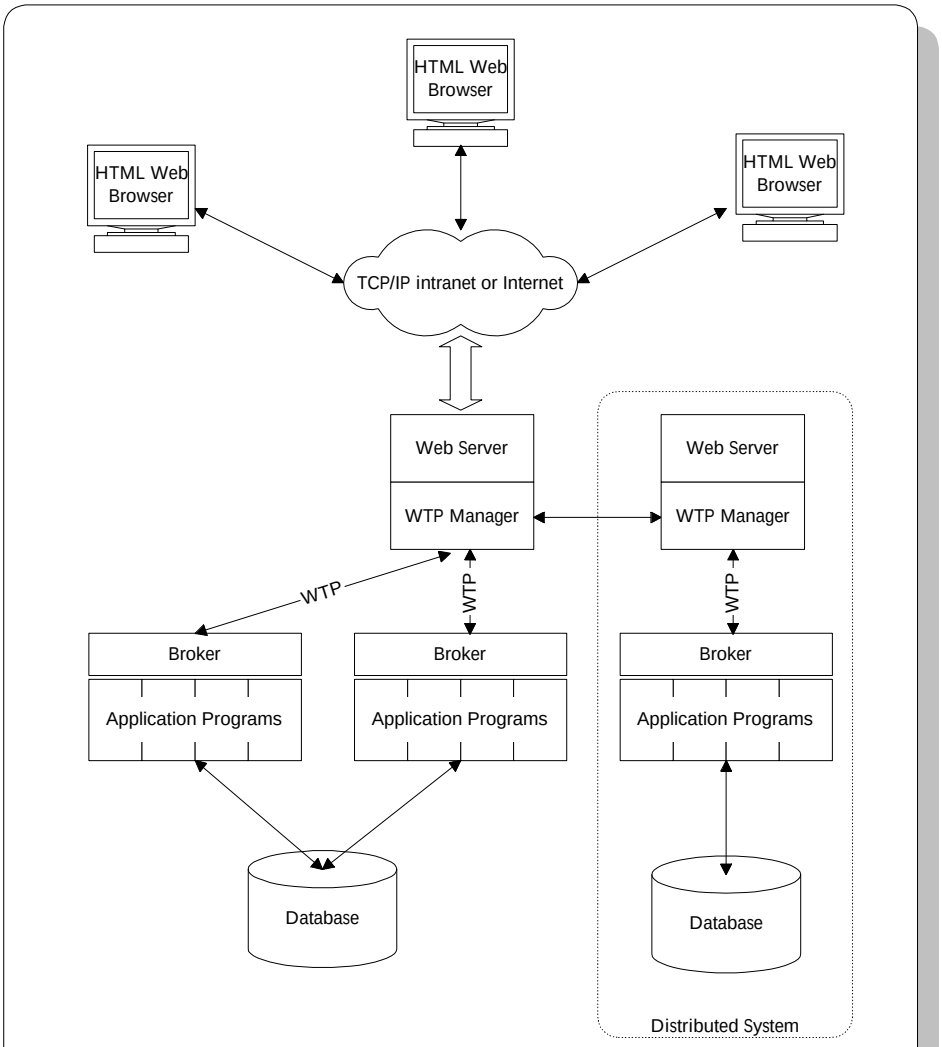


Fig 2. General Operating Principles

There are four main components:

1. The web browser, which provides the screen I/O layer.

2. The web server and WTP manager. This is a gateway between the web HTTP protocol and the iMatix WTP (web transaction protocol). Any appropriate server can be used: the iMatix Web Transaction Server is one solution, but others are possible.

3. The broker. This is a generic layer that connects specific application programs to the WTP manager.

4. The application program. This is a program that handles functional aspects of the application, such as display or update of some data.

The broker is by design on the same system as the WTP manager. However, iMatix Studio will in the near future support transparently distributed applications using a peer-to-peer protocol between distributed WTP managers.

4.1 The Screen I/O Layer

iMatix Studio is a transaction-processing system. This means that a limited number of executable programs handle transactions collected from multiple client systems. Transaction processing systems are efficient and good at handling database access.

We use a forms metaphor to collect data for processing. The user receives a form, fills-it in, and selects some action. The application program processes the entire form in one go. This is quite different from traditional Windows-type programming, where the application program processes each data field or action as it occurs. It is our experience that the form-based approach is simpler and thus cheaper.

The standard web user-interface is generally excellent, but suffers in some aspects:

1. It is difficult to work without a mouse: the keyboard is not well-used in most browsers. For example: the cursor keys do not work usefully.

2. The form interface is generally poor: for example input fields can only be shown one manner; there is no easy way to distinguish normal input fields from erroneous input fields or important input fields.

In contrast, the Web user-interface is excellent in many other areas, notably integration with document presentation, images, hyper-links, fonts, tables, scrolling, etc.

iMatix Studio works well with standard web browsers such as MSIE and Netscape Navigator, and uses small amounts of JavaScript to improve aspects of the user interface. We are also working with a European firm to develop a version of their browser that will provide an excellent level of ergonomics for keyboard-intensive work, as well as running 2-3 times faster than either of the mainstream browsers.

4.2 The Web Server and WTP Manager

The web server manages connections to the multiple browsers which access the application; the WTP manager handles the transaction-processing requirements of the application. In general these two are combined into a single process, although other configurations are possible.

In general one WTP manager can handle multiple applications at once; each application is accessed using a different URL (universal resource locator, e.g. `http://www.imatix.com/wtp/welcome`).

When starting-up an application, the WTP manager launches, then creates connections with a number of *brokers.* Each broker registers a number of *application programs* with the WTP manager. When the WTP manager needs to call a specific program, it will find an appropriate and available broker (there may be several) and pass the request to that broker.

The WTP manager knows when a new user connects to the application. It is responsible for handling a user session, from beginning to end.

The WTP manager handles:

- Broker process management - starting, stopping, and monitoring brokers.
- Program dispatching - choosing an available broker, and sending a transaction request to that broker.
- Session control - creating new sessions, and ending old or finished sessions.
- Context management - storing application program context (memory) between transactions.

4.3 The Broker

In order to encapsulate the WTP interface, we use a *broker* program. This program assures the connection between the WTP manager, and the application programs:

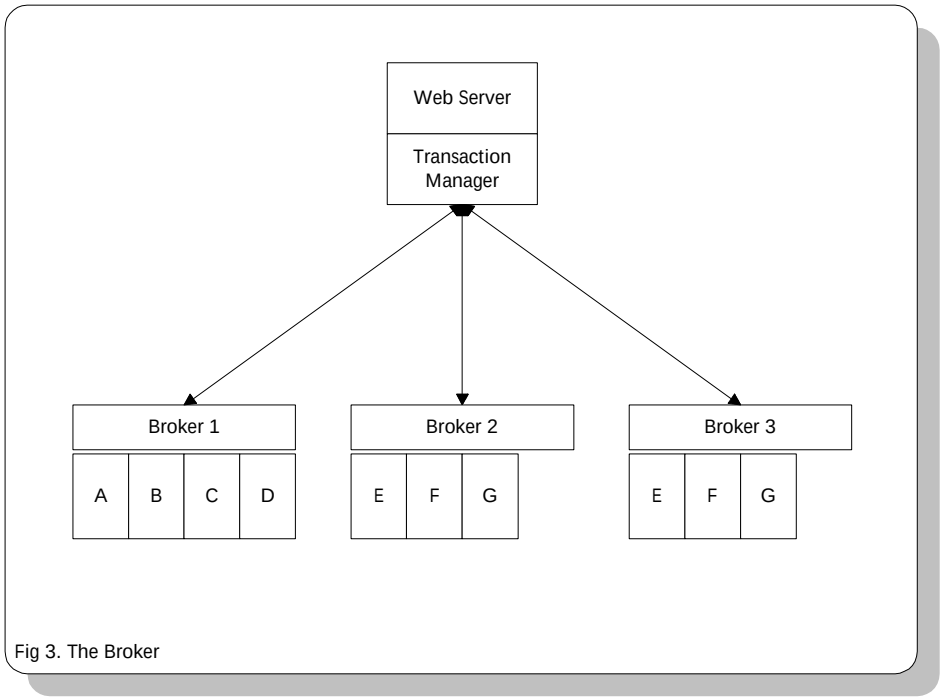


Fig 3. The Broker

The broker works as follows:

1. At start-up, it connects to the WTP manager and announces the list of programs that it handles. We call this the *registration* process. For example, when broker 1 starts-up, it registers programs A, B, C, and D. The actual connection is made using a local TCP/IP

socket, based on information that the WTP manager passes to the broker when it invokes it.

2. When the broker receives a transaction-processing request, it passes it to the appropriate application program. When the application program ends processing, the broker sends the response back to the WTP manager.

We gain much by using this technique, rather than handling the WTP protocol directly in the application programs:

- The application programs are independent of the WTP manager. For example, they can run without the web server, in a much simpler environment, for testing, or under another gateway protocol such as CGI,[3] It can be integrated into different architectures without modification.

- Application programs can be linked into executable programs in any manner. For example: to link a Oracle Pro*C program[4] into an executable, one adds a megabyte or so of Oracle runtime functions. When linking several Pro*C programs together, the runtime is only included once.

- Several instances of a broker (with its programs) can be started. Since transaction processing is serially-re-entrant, two users cannot start the same program at the same time unless two copies are running. By starting several instances, the application manager can choose to improve response time for specific programs.

- Application programs can be grouped in different ways to allow *load-balancing*. Typically, heavily-used (often large) programs are linked separately, while little-used programs may be linked together.

- Brokers can be started and stopped on-the-fly.

A typical large application could consist of 500-2000 programs, split over 10-20 separate brokers. Broker programs are generated from a standard template, since the bulk of the broker code is totally standard.

4.4 The Application Program

While it is possible to write brokers in any language that supports TCP/IP sockets, and using any approach, iMatix Studio provide a standard development model which can be applied equally to any language. The current implementation is in C; we are developing support for C++, Java, COBOL, and Perl.

Basically, a Studio application program is built using the iMatix **Libero** tool; this uses a finite-state machine methodology to generate code from a high-level definition of the program's logic. This sounds complex, but is actually straight-forward, and an elegant way

[3] iMatix Studio supports CGI, and Studio applications can run under CGI with any web server. WTP is much faster, however, for reasons explained in Appendix C.

[4] I.e. a C program that does Oracle SQL calls.

to work. This model is not obligatory, and in some cases it would not be appropriate. However, it does work exceedingly well for normal interactive application programs.

This figure shows how an application program is composed of a number of layers; the shaded boxes show generated code:

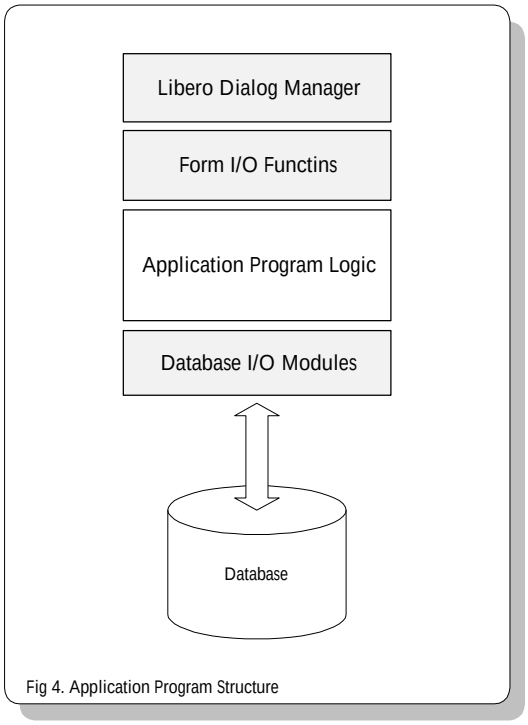


Fig 4. Application Program Structure

- The **Libero Dialog Manager** is responsible for the execution of the application program. This is an include file which contains the compiled tables which drive the program.
- The **Form I/O functions** generate the HTML required by the program, depending the form data it wants to display.
- The **Application program logic** consists of a number of modules of business logic; typically these will check and calculate data, and move it between the database and the form.
- The **Database I/O functions** are hand-written or generated modules that handle all access to the database. This layer is optional; it simply makes it possible to change databases, cheaply.

4.4.1 The Libero Dialog Manager

The generated dialog manager plays a critical role in a Studio application program. It provides a formal framework for writing robust code, and also abstracts a specific control layer of great importance to transactional systems. A transaction processing system basically works in a series of steps: show some information to the user, allow input of data, accept an action code, execute the action, and show some new information. The generated dialog manager code encapsulates these steps, so that the program developer can concentrate on writing the business logic.

A more detailed explanation of Libero is given in the next chapter.

4.4.2 The Form I/O Functions

The Studio toolkit includes form design and code generation tools. These simplify the normally tedious process of producing the correct HTML for displaying a form, and decoding the form data afterwards. Rather than work at the HTML level, the program developer works with an abstracted form object with properties such as a list of data fields. iMatix Studio generates the HTML for the form dynamically at run-time. We designed an abstraction that handles most of the needs of business applications, without being too complex. At the same time, the developer can work at lower levels, including in HTML directly if required.

This approach makes form design and reuse a simple, rapid process, yet gives full control for cases where it's necessary. A more detailed explanation of the Studio form I/O system is given in the next chapter.

4.4.3 The Application Program Logic

The application program logic consists of a set of hand-written business logic modules, following the structure of the dialog. In many cases, these modules are shared between programs, and in some cases, entire programs can be generated from templates. We plan to provide program generation tools in iMatix Studio at a later date.

The language chosen is fairly arbitrary: with the program design done using Libero, it is a matter of preference for a project. Studio applications can also be built out of a mix of languages, for instance with some programs written in C, and some in COBOL.

4.4.4 The Database I/O Modules

Our experience with Oracle, IBM DB2, Informix, Ingres, Digital RDB, ODBC, and other databases and interfaces allows us to fully understand the importance, requirements and limitations of this layer. A 'naïve' developer will embed calls to the database directly in their program. We recommend, and assist, but don't enforce, a packaging of such database access code into subroutines for a number of reasons:

1. When the database access code is placed in an identifiable set of subroutines, it can be corrected, tuned, and maintained at less cost than if it is mixed with the application business logic.

2. In a majority of cases, the database access code can be generated from templates. This does require a more formal approach (usually data-dictionary based), but opens the door to an important set of possibilities - database independence, richer error handling, portability.

3. The database access code is one of the areas where portability is compromised. For instance, an application developed using ODBC calls cannot be ported transparently to native database calls, when performance becomes an issue. If the database access has been separated from the application code, this becomes a more realistic proposition.

4. It becomes possible to employ specialists to write the database access code, which can save development time, lower costs, and improve quality.

To demonstrate the consequences of such an approach: we participated in the development of a large tour operator system used in several European countries by large tour operators. This application is portable, and has been deployed on IBM MVS/CICS, Digital ACMS, and Unix. The database access code was written as a set of access modules (generally one per table or join), and could be moved, by a single person, to a new database (e.g. from IBM DB2 to Oracle or Informix) with four to six weeks of work.

5. iMatix Studio Implementation

This chapter provides an in-depth technical description of how iMatix Studio was implemented, and how a typical application would be developed. This information is not necessary to understand the benefits of iMatix Studio, and the non-technical reader may wish to skip ahead to page 40.

5.1 Technical Basis For iMatix Studio

The technical foundations for iMatix Studio were the various open technologies already developed by iMatix:

- The Libero development environment;
- The SFL portability library, which provides a platform for portable C applications;
- The SMT multithreading kernel, which provides a platform for high-performance internet servers;
- The htmlpp HTML pre-processor, which provides a macro generator for HTML pages.

These technologies are stable, well-tested, and richly-documented. They have been used by iMatix and numerous others in mission-critical projects since 1992. They are provided as free software on the Internet. The sources are freely available, and written so as to be easy to understand and modify if necessary. This strategy was designed to establish a solid and rich technical foundation that was also open and accessible.

5.2 The iMatix Web Transaction Server

The iMatix Web Transaction Server is based on our widely-used *Xitami* web server. The software is built as a set of component agents which communicate using the SMT kernel:

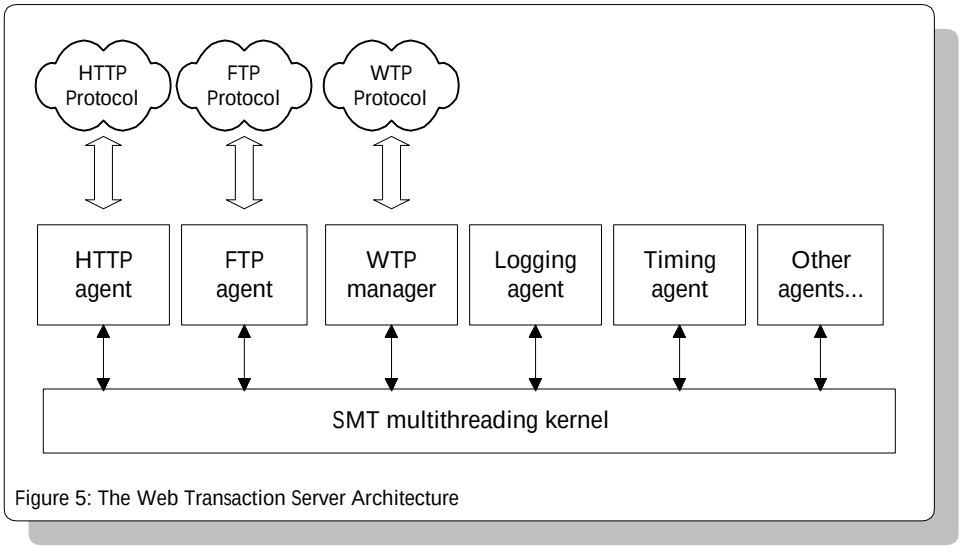


Figure 5: The Web Transaction Server Architecture

We used the SMT kernel for its speed and stability; the reliability of a web application is only as good as its web server. The iMatix Web Transaction Server will run without maintenance for long periods of time, and can handle many hundreds of simultaneous connections thanks to the light multithreading model provided by SMT.

5.3 The Web Transaction Protocol (WTP)

In order to define a clear interface between the application programs and the web server, we designed a protocol that could be implemented by anyone, and in any web server. The WTP protocol is described in Appendix C. WTP can be used for other purposes than iMatix Studio; this is simply one possible use. In general, WTP can be seen as a replacement for the Common Gateway Interface (CGI) which is often used for web application development. The main improvements that WTP offers are:

1. It improves response time by keeping application programs in memory;

2. It supports applications consisting of multiple programs;

3. It handles program context management;

4. It allows load-balancing by distributing transaction requests between multiple instances of an application program;

5. It allows transparent distribution of an application across multiple platforms.

The WTP definition assumes a reference implementation; this is provided by the iMatix Web Transaction Server.

5.4 Building a Studio Application

5.4.1 The Studio Form I/O System

5.4.1.1 Design Goals and Objectives

HTML is a good language for mark-up (i.e. defining information layout and representation), but is a clumsy way to define forms. Any developer of web applications (e.g. using CGI) will know and recognise the same set of problems:

- A user-interface designer has to be expert in HTML, HTTP, and JavaScript.
- Since the HTML tags are emitted by the program, using embedded **print** statements, it is very hard to separate user-interface design from program design.
- For the same reason, forms are generally simplistic, clumsy, and extraordinarily expensive to maintain.

A HTML editor like MS FrontPage can solve the first problem - a UI designer can create a form without needing to be a programmer. However, the program developer must then re-engineer this as source code.

The main objective of the form I/O system is to eliminate these problems. We do this by:

- Creating the concept of a *form* as a concrete, external object.
- Providing the means for the user-interface designer to work without requiring a running application. (I.e. a preview of the form's HTML code.)
- Providing the user-interface designer with an abstraction that makes form design faster. Specifically, using a high-level abstraction of common elements such as text fields, select list, etc., that automatically produces the necessary HTML.
- Providing the UI designer with access to the HTML level if necessary, when full control over layout is needed.
- Creating a development cycle that allows rapid prototyping of forms, and independent refinement of the form and the program.

5.4.1.2 Architecture of the Form I/O System

The Studio form I/O system is a set of code generators and run-time functions. These work as shown in the following figure:

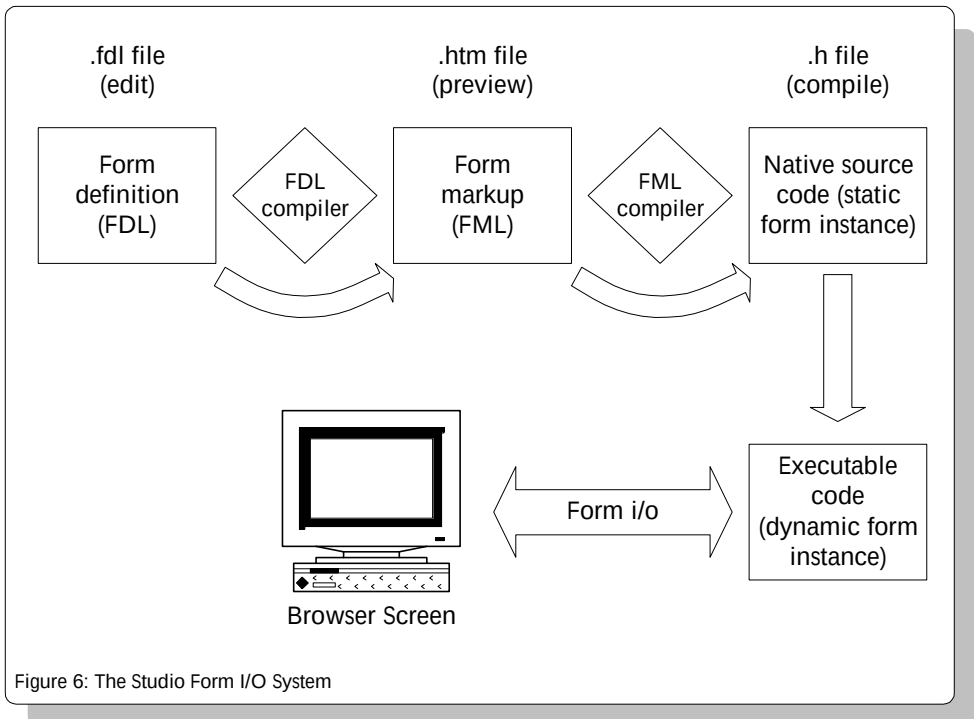


Figure 6: The Studio Form I/O System

A form description is a mix of three languages:

1. The HTML tags which manage the layout of the information on the screen.

2. A set of form markup language (FML) tags, which look like HTML comments. Each FML tag defines a field on the form. FML is compatible with HTML so that a text file containing only HTML and FML can be displayed in a browser.

3. A set of form definition language (FDL) instructions. These are macros that create fields and tables of varying types. FDL is *not* compatible with HTML.

5.4.1.3 Example of a Web Form

We'll look at an example of a simple form definition that asks for some input. The starting point for the form is a .fdl file that looks like this:

```
.include macros.def
.block header
<HTML><BODY><HR>
.block footer
<HR>
.image html/im0096c.gif iMatix Corporation
</BODY></HTML>
.end

.page example = "Example form"
.fields
.textual "Your name:"    user-name    size=20 max=50
.numeric "Your age:"     user-age     size=3
.radio   "Your species:" user-species 1=Human -
                                      2=Martian 3=Other
.endfields
```

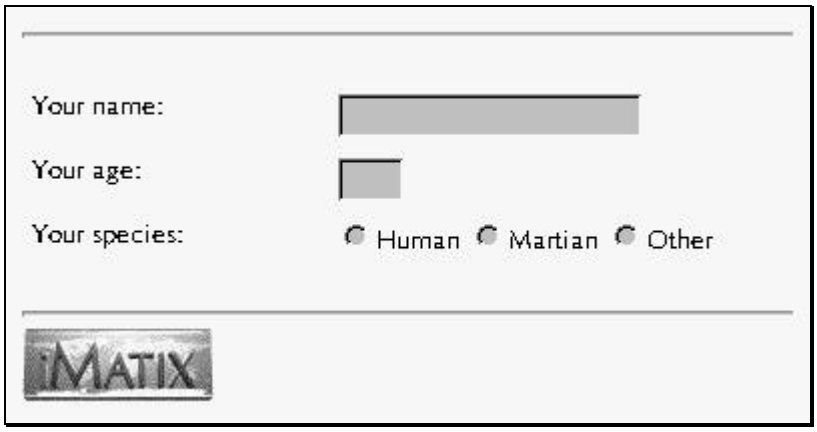
The FDL file is compiled by fdlgen, which creates an HTML file containing the resulting mix of HTML and FML tags. A single FDL action, like '.textual' will result in a series of HTML and FML tags.

```
$ fdlgen example

fdlgen - Studio FDL compiler V2.0
Copyright (c) 1996-98 iMatix - http://www.imatix.com

fdlgen I: processing example.fdl...
```

This is the HTML file generated by fdlgen:

```
<HTML><BODY><HR>
<FORM METHOD=POST ACTION="#(uri)">
<INPUT TYPE=HIDDEN NAME=jsaction VALUE="">
<TABLE WIDTH=750><TR><TD ALIGN=LEFT VALIGN=TOP NOWRAP>
<!--FML FIELD=F1 TEXTUAL LABEL NAME=L_user-name VALUE="Your name:" -->
Your name:
<!--FML /FIELD-->
</TD><TD ALIGN=LEFT NOWRAP WIDTH=80%>
<!--FML FIELD=F2 TEXTUAL INPUT NAME=user-name
SIZE=20 MAX=50 VALUE="" -->
<INPUT TYPE=TEXT SIZE=20 VALUE="">
<!--FML /FIELD-->
</TD></TR><TR><TD ALIGN=LEFT VALIGN=TOP NOWRAP>
<!--FML FIELD=F3 TEXTUAL LABEL NAME=L_user-age VALUE="Your age:" -->
Your age:
<!--FML /FIELD-->
</TD><TD ALIGN=LEFT NOWRAP WIDTH=80%>
<!--FML FIELD=F4 NUMERIC INPUT NAME=user-age
SIGN=? DECS=0 DECFMT=? FILL=? BLANK=0
COMMA=0 SIZE=3 MAX=? VALUE="" -->
<INPUT TYPE=TEXT SIZE=3 VALUE="">
<!--FML /FIELD-->
</TD></TR><TR><TD ALIGN=LEFT VALIGN=TOP NOWRAP>
<!--FML FIELD=F5 TEXTUAL LABEL NAME=L_user-species VALUE="Your species:" -->
Your species:
<!--FML /FIELD-->
</TD><TD ALIGN=LEFT NOWRAP WIDTH=80%>
<!--FML FIELD=F6 RADIO INPUT NAME=user-species
COLUMN=0 DETAIL=0 VALUE=0 NULL="?" -->
<!--FML /FIELD-->
<!--FML FIELD=F6 RADIO OPTION="Human" -->
<INPUT TYPE=RADIO NAME=user-species>Human
<!--FML /FIELD-->
<!--FML FIELD=F6 RADIO OPTION="Martian" -->
<INPUT TYPE=RADIO NAME=user-species>Martian
<!--FML /FIELD-->
<!--FML FIELD=F6 RADIO OPTION="Other" -->
<INPUT TYPE=RADIO NAME=user-species>Other
<!--FML /FIELD-->
</TD></TR></TABLE>
</FORM><HR>
<IMG SRC="html/im0096c.gif" WIDTH="96"
HEIGHT="36" ALT="iMatix Corporation">
</BODY></HTML>
```

Immediately we see why it's desirable to hide this. The HTML file is not meant for human consumption; rather it allows the UI designer to immediately view the form in a browser:



The HTML/FML file is compiled by fmlgen, which creates a C language include file containing data structures and definitions that are compiled with the program source code.

```
$ fmlgen example

fmlgen - Studio form code generator V2.0
Copyright (c) 1996-98 iMatix - http://www.imatix.com

fmlgen I: processing example.htm...
```

This is the C include file generated by fmlgen:

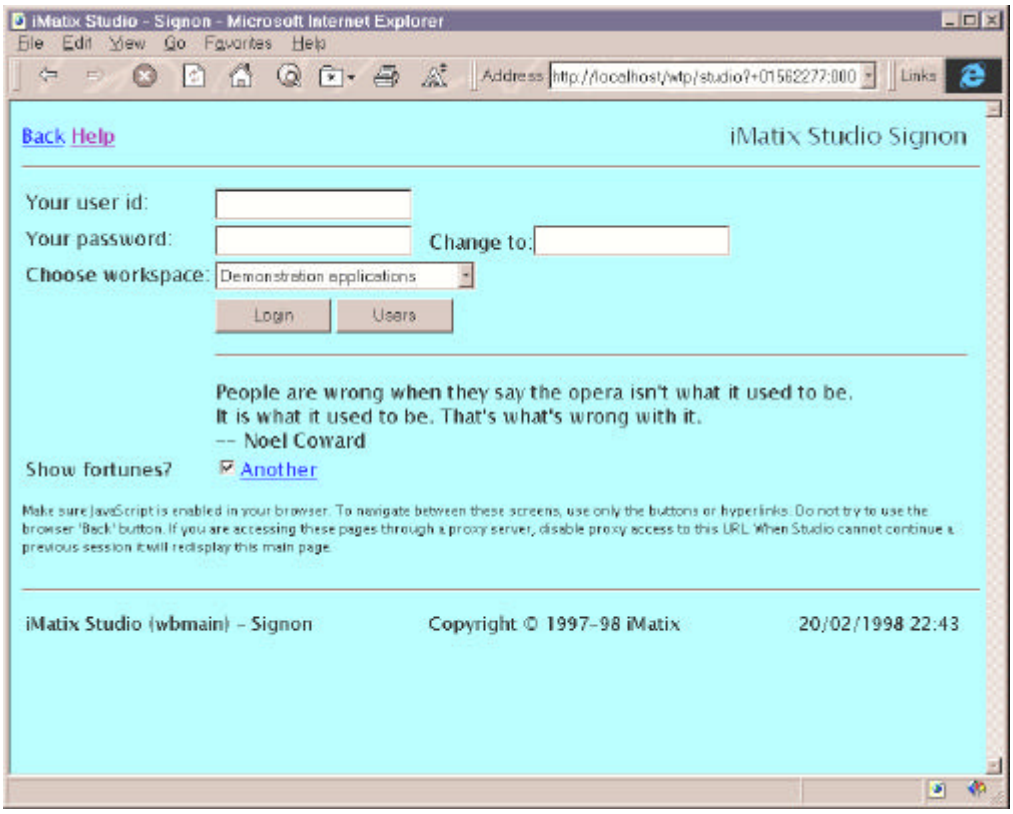
```
/*---------------------------------------------------------------------------
 *  example.h - HTML form definition
 *
 *  Generated 1998/02/16, 16:39:15 by fxgen 2.0
 *---------------------------------------------------------------------------*/

#ifndef __FORM_EXAMPLE__
#define __FORM_EXAMPLE__

#include "sfl.h"
#include "formio.h"

/*  Constants defining size of tables, etc.                                  */

#define EXAMPLE_L_USER_NAME                 0
#define EXAMPLE_USER_NAME                   1
#define EXAMPLE_L_USER_AGE                  2
#define EXAMPLE_USER_AGE                    3
#define EXAMPLE_L_USER_SPECIES              4
#define EXAMPLE_USER_SPECIES                5

/*  This table contains each block in the form                               */

static byte example_blocks [] = {
    /*  <HTML><BODY><HR>                                                     */
    0, 17, 0, '<', 'H', 'T', 'M', 'L', '>', '<', 'B', 'O', 'D', 'Y',
    '>', '<', 'H', 'R', '>',
    /*  <FORM METHOD=POST ACTION="#(uri)">                                   */
    0, 35, 0, '<', 'F', 'O', 'R', 'M', 32, 'M', 'E', 'T', 'H', 'O', 'D',
    '=', 'P', 'O', 'S', 'T', 32, 'A', 'C', 'T', 'I', 'O', 'N', '=', '"',
    '#', '(', 'u', 'r', 'i', ')', '"', '>',
    /*  <INPUT TYPE=HIDDEN NAME=jsaction VALUE="">                           */
    0, 43, 0, '<', 'I', 'N', 'P', 'U', 'T', 32, 'T', 'Y', 'P', 'E', '=',
    'H', 'I', 'D', 'D', 'E', 'N', 32, 'N', 'A', 'M', 'E', '=', 'j', 's',
    'a', 'c', 't', 'i', 'o', 'n', 32, 'V', 'A', 'L', 'U', 'E', '=', '"',
    '"', '>',
    /*  <TABLE WIDTH=750>                                                    */
    0, 18, 0, '<', 'T', 'A', 'B', 'L', 'E', 32, 'W', 'I', 'D', 'T', 'H',
    '=', '7', '5', '0', '>',
    /*  <TR><TD ALIGN=LEFT VALIGN=TOP NOWRAP>                                */
    0, 38, 0, '<', 'T', 'R', '>', '<', 'T', 'D', 32, 'A', 'L', 'I', 'G',
    'N', '=', 'L', 'E', 'F', 'T', 32, 'V', 'A', 'L', 'I', 'G', 'N', '=',
    'T', 'O', 'P', 32, 'N', 'O', 'W', 'R', 'A', 'P', '>',
```

```
    /*  !--FIELD TEXTUAL f1 NAME=L_user-name VALUE="Your name:"             */
    0, 21, 10, 6, 1, 0, 10, 0, 10, 'f', '1', 0, 'Y', 'o', 'u', 'r', 32,
    'n', 'a', 'm', 'e', ':', 0,
    /*  </TD><TD ALIGN=LEFT NOWRAP WIDTH=80%>                              */
    0, 38, 0, '<', '/', 'T', 'D', '>', '<', 'T', 'D', 32, 'A', 'L', 'I',
    'G', 'N', '=', 'L', 'E', 'F', 'T', 32, 'N', 'O', 'W', 'R', 'A', 'P',
    32, 'W', 'I', 'D', 'T', 'H', '=', '8', '0', '%', '>',
    /*  !--FIELD TEXTUAL f2 NAME=user-name SIZE=20 MAX=50 VALUE=""          */
    0, 11, 10, 0, 1, 0, 20, 0, '2', 'f', '2', 0, 0,
    /*  </TD></TR>                                                         */
    0, 11, 0, '<', '/', 'T', 'D', '>', '<', '/', 'T', 'R', '>',
    /*  <TR><TD ALIGN=LEFT VALIGN=TOP NOWRAP>                              */
    0, 4, 1, 0, 0, 'y',
    /*  !--FIELD TEXTUAL f3 NAME=L_user-age VALUE="Your age:"              */
    0, 20, 10, 6, 1, 0, 9, 0, 9, 'f', '3', 0, 'Y', 'o', 'u', 'r', 32,
    'a', 'g', 'e', ':', 0,
    /*  </TD><TD ALIGN=LEFT NOWRAP WIDTH=80%>                              */
    0, 4, 1, 0, 0, 184,
    /*  !--FIELD NUMERIC f4 NAME=use ... MMA=0 SIZE=3 MAX=? VALUE=""       */
    0, 17, 11, 0, 1, 0, 3, 0, 3, 0, 0, 0, 0, 0, 0, 'f', '4', 0, 0,
    /*  </TD></TR>                                                         */
    0, 4, 1, 0, 0, 237,
    /*  <TR><TD ALIGN=LEFT VALIGN=TOP NOWRAP>                              */
    0, 4, 1, 0, 0, 'y',
    /*  !--FIELD TEXTUAL f5 NAME=L_user-species VALUE="Your species:"      */
    0, 24, 10, 6, 1, 0, 13, 0, 13, 'f', '5', 0, 'Y', 'o', 'u', 'r', 32,
    's', 'p', 'e', 'c', 'i', 'e', 's', ':', 0,
    /*  </TD><TD ALIGN=LEFT NOWRAP WIDTH=80%>                              */
    0, 4, 1, 0, 0, 184,
    /*  !--FIELD RADIO f6 NAME=user- ... 0 DETAIL=0 VALUE=0 NULL="?"       */
    0, 23, 16, 0, 1, 0, 0, 'f', '6', 0, '0', 0, 'N', 'o', 32, 's', 'e',
    'l', 'e', 'c', 't', 'i', 'o', 'n', 0,
    /*  !--FIELD RADIO f6 OPTION="Human"                                   */
    0, 10, 17, 1, '[', 1, 'H', 'u', 'm', 'a', 'n', 0,
    /*  !--FIELD RADIO f6 OPTION="Martian"                                 */
    0, 12, 17, 1, '[', 2, 'M', 'a', 'r', 't', 'i', 'a', 'n', 0,
    /*  !--FIELD RADIO f6 OPTION="Other"                                   */
    0, 10, 17, 1, '[', 3, 'O', 't', 'h', 'e', 'r', 0,
    /*  </TD></TR>                                                         */
    0, 4, 1, 0, 0, 237,
    /*  </TABLE>                                                           */
    0, 9, 0, '<', '/', 'T', 'A', 'B', 'L', 'E', '>',
    /*  </FORM>                                                            */
    0, 8, 0, '<', '/', 'F', 'O', 'R', 'M', '>',
    /*  <HR>                                                               */
    0, 5, 0, '<', 'H', 'R', '>',
    /*  <IMG SRC="html/im0096c.gif" WIDTH="96"                             */
    0, 39, 0, '<', 'I', 'M', 'G', 32, 'S', 'R', 'C', '=', '"', 'h', 't',
    'm', 'l', '/', 'i', 'm', '0', '0', '9', '6', 'c', '.', 'g', 'i',
    'f', '"', 32, 'W', 'I', 'D', 'T', 'H', '=', '"', '9', '6', '"',
    /*  HEIGHT="36" ALT="">                                                */
    0, 20, 0, 'H', 'E', 'I', 'G', 'H', 'T', '=', '"', '3', '6', '"', 32,
    'A', 'L', 'T', '=', '"', '"', '>',
    /*  </BODY></HTML>                                                     */
    0, 15, 0, '<', '/', 'B', 'O', 'D', 'Y', '>', '<', '/', 'H', 'T',
    'M', 'L', '>',
    0, 0, 0
    };

static FIELD_DEFN example_fields [] = {
    { 0, 161, 10 },                     /*  l_user_name                    */
    { 12, 224, 50 },                    /*  user_name                      */
    { 64, 256, 9 },                     /*  l_user_age                     */
    { 75, 284, 3 },                     /*  user_age                       */
    { 80, 315, 13 },                    /*  l_user_species                 */
    { 95, 347, 3 },                     /*  user_species                   */
    { 100, 0, 0 },                      /*  -- sentinel --                 */
    };

/*  The data of a form is a list of attributes and fields                  */

typedef struct {
    byte   l_user_name_a        ;
    char   l_user_name          [10 + 1];
    byte   user_name_a          ;
    char   user_name            [50 + 1];
    byte   l_user_age_a         ;
    char   l_user_age           [9 + 1];
    byte   user_age_a           ;
    char   user_age             [3 + 1];
    byte   l_user_species_a     ;
    char   l_user_species       [13 + 1];
    byte   user_species_a       ;
    char   user_species         [3 + 1];
    } EXAMPLE_DATA;
```

```
/*  The form definition collects these tables into a header                 */

static FORM_DEFN form_example = {
    example_blocks,
    example_fields,
    31,                                 /*  Number of blocks in form        */
    6,                                  /*  Number of fields in form        */
    0,                                  /*  Number of actions in form       */
    100,                                /*  Size of fields                  */
    "example",                          /*  Name of form                    */
    };

#endif                                  /*  End included file               */
```

5.4.1.4 Other Examples of Web Forms

This screen shot shows the sign-on screen for the iMatix Studio Workbench application:

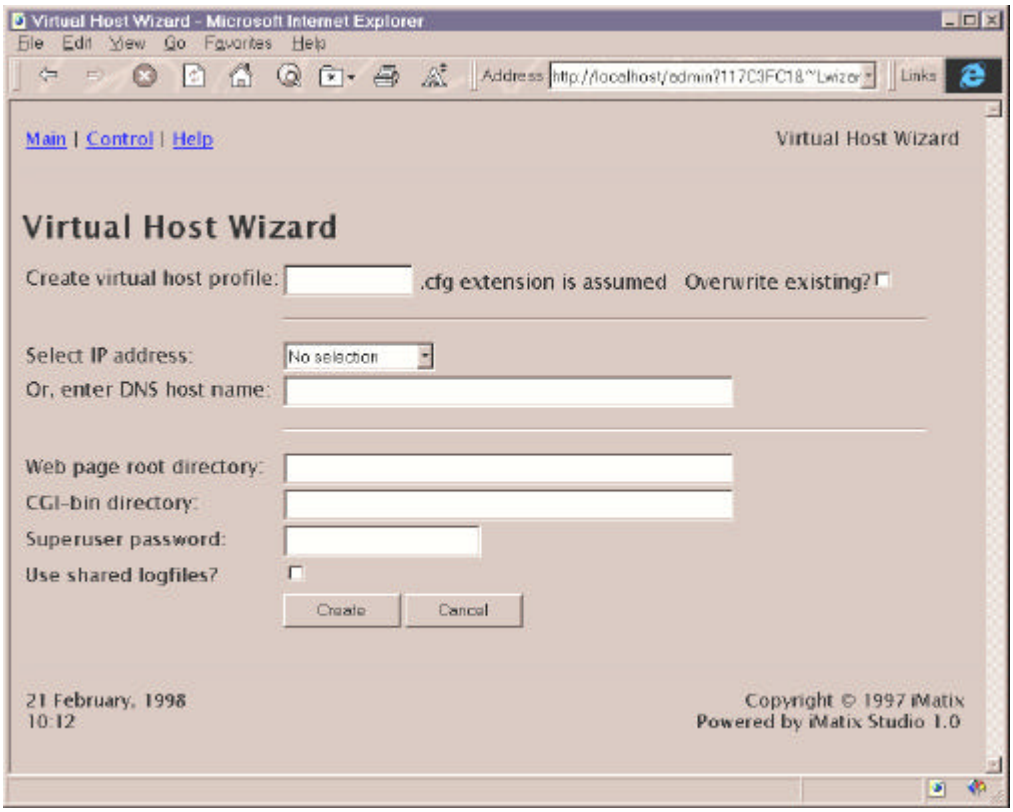


This is the FDL definition for the form:

```
.include prelude.def
.page wbmain = "Signon"
.title_bar iMatix Studio Signon
.fields
.textual "Your user id:"     user-id      size=$(USER_ID_MAX)
.textual "Your password:"    password     size=$(PASSWORD_MAX) attr=secure
.textual "Change to:"        new-password size=$(PASSWORD_MAX) attr=secure -
                                          join=yes
.select  "Choose workspace:" workspaces   type=dynamic
<BR>
.action "" Login     event=login_event
.action "" Apply     event=apply_event    join=yes
.action "" Users     event=users_event    join=yes
.do if show-fortune
<HR>
.textbox "" fortune rows=16 cols=80 attr=label
.enddo
.boolean "Show fortunes?"    show-fortune
.do if show-fortune
.action "" Another   event=another_event  join=yes type=plain
.enddo
.endfields
```

```
<P><FONT SIZE=-2>
Make sure JavaScript is enabled in your browser.
To navigate between these screens, use only the buttons or hyperlinks. Do not
try to use the browser 'Back' button. If you are accessing these pages through
a proxy server, disable proxy access to this URL. When Studio cannot continue
a previous session it will redisplay this main page.
```

This screen shot shows the Virtual Host Wizard from the Xitami web server administration application:



This is the FDL definition for the form (the definitions for the page header and footer are not shown):

```
.page xiadm25 = "Virtual Host Wizard"
<HR><H2>$(TITLE)</H2>

.fields
.textual "Create virtual host profile:"   host-file  size=12 max=$(FNSIZE) -
    notes=".cfg extension is assumed"
.boolean "Overwrite existing?"            overwrite  join=yes
<HR>
.select  "Select IP address:"             host-addr  type=dynamic
.textual "Or, enter DNS host name:"       host-name  size=50
<HR>
.textual "Web page root directory:"       webpages   size=50
.textual "CGI-bin directory:"             cgi-bin    size=50
.textual "Superuser password:"            superuser  size=20
.boolean "Uses shared logfiles?"          sharelogs
.boolean "Can use browser-based Admin?"   use-admin  value=no
.textual "If so, user id:"                admin-user size=20
.textual "password:"                      admin-pass size=20 join=yes
<P>
.label  ""
.action "" Create     event=define_event   join=yes
.action "" Cancel     event=cancel_event   join=yes
.endfields
```

5.4.1.5 Advantages of the Studio Form I/O System

1. Web forms are easy to prototype, design, maintain, and use in application programs.

2. The form is independent of the programming language.

3. The UI designer can describe the form in terms of high-level concepts, rather than in low-level HTML tags.

4. The developer can still use low-level tags for explicit layout control when required.

5. The form I/O runtime provides a rich ergonomy on all forms, including: date and numeric field checking, dynamic field types (e.g. switching between input and output), automatic use of JavaScript when required, etc.

5.4.2 The Studio Database I/O System

5.4.2.1 Design Goals and Objectives

The Studio database I/O system has these design goals and objectives:

1. To simplify the work involved in accessing data in a relational database, in ISAM (indexed) files, or in other persistent data structures.

2. To isolate such code so that it can be developed, tuned, and maintained by specialists, instead of requiring that all application developers be expert in the database system.

3. To provide a means by which the application can be rendered independent of the specific database; for instance it can be portable to any ISAM support, or to any relational database.

4. To generate such code where possible, thus ensuring that the highest quality can be achieved at the lowest cost.

5.4.2.2 Architecture of the Database I/O System

The Studio database I/O system is a set of database schemas and code generators. These work as shown in the following figure:

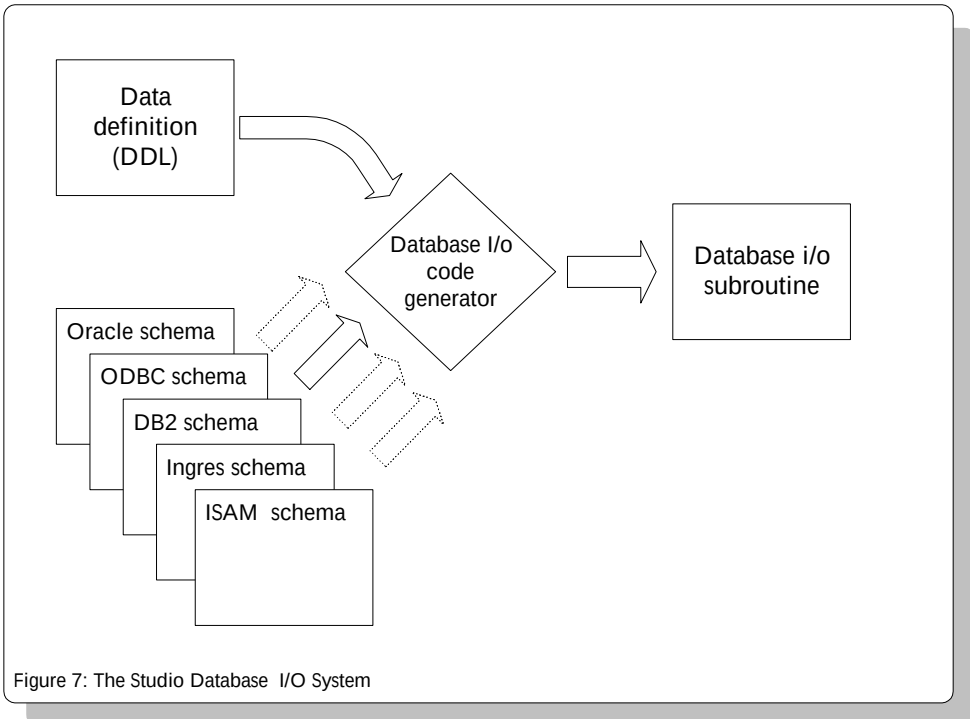


Figure 7: The Studio Database I/O System

Each generated database I/O subroutine handles a set of standard accessed for a single table. Typically, such a subroutine will provide read and update access to the table using the main table keys.

The majority of database I/O routines can be generated, and are thus built at little or no cost; there will generally be a set of cases (perhaps 10% of the total) where such generic access is too slow or clumsy, and where hand-written SQL is required.

5.4.3 The Libero Tool

iMatix Studio makes heavy use of Libero as a method for program design, abstraction, and transaction control. Libero was used in building the iMatix Web Transaction Server, to build the various design and code generation tools, and to abstract the structure of a Studio application program.

Our heavy use of Libero comes from considerable positive experience with this tool and related tools. We have seen that it is widely applicable, simple to learn, robust, and beneficial.

5.4.3.1 Overview of The Libero Development Environment

This figure shows how a program developer uses Libero:

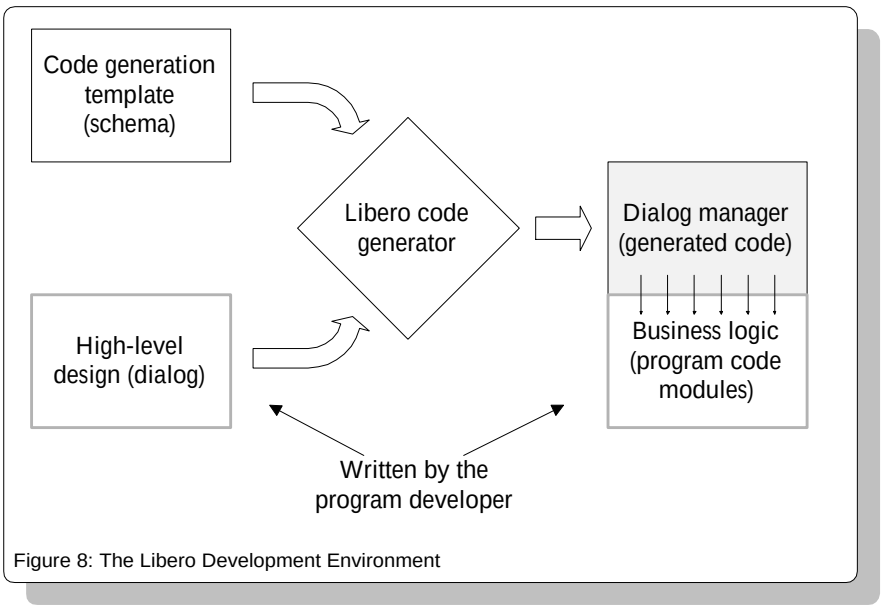


Figure 8: The Libero Development Environment

There are code generation templates (also called 'schemas') for various purposes and programming languages. For instance, iMatix Studio provides a standard schema for application programs written in C. The schema is a script that tells the Libero code generator exactly what code should be generated. There are many schemas provided with Libero; for instance for writing functions in other languages, for developing shell scripts, Perl tools, and so on. The iMatix Studio schema is just one example. Libero schema can be (and often are) tuned according to the specific needs of a project.

The basic development cycle is this:

1. The developer writes a dialog that describes the program's logic.

2. The Libero code generator produces a dialog manager that implements the specific dialog.

3. At the same time, Libero creates empty stubs for any new modules that are referenced in the dialog. If necessary, it will first create a new, skeleton program, again as instructed by the schema.

4. The developer completes some or all of the modules, compiles, and tests the program. To further refine the program, the developer continues working in the dialog, or in the program modules.

This process is fully circular, and allows rapid prototyping with a smooth transition to a completed program.

5.4.3.2 Example of Writing a Program Using Libero

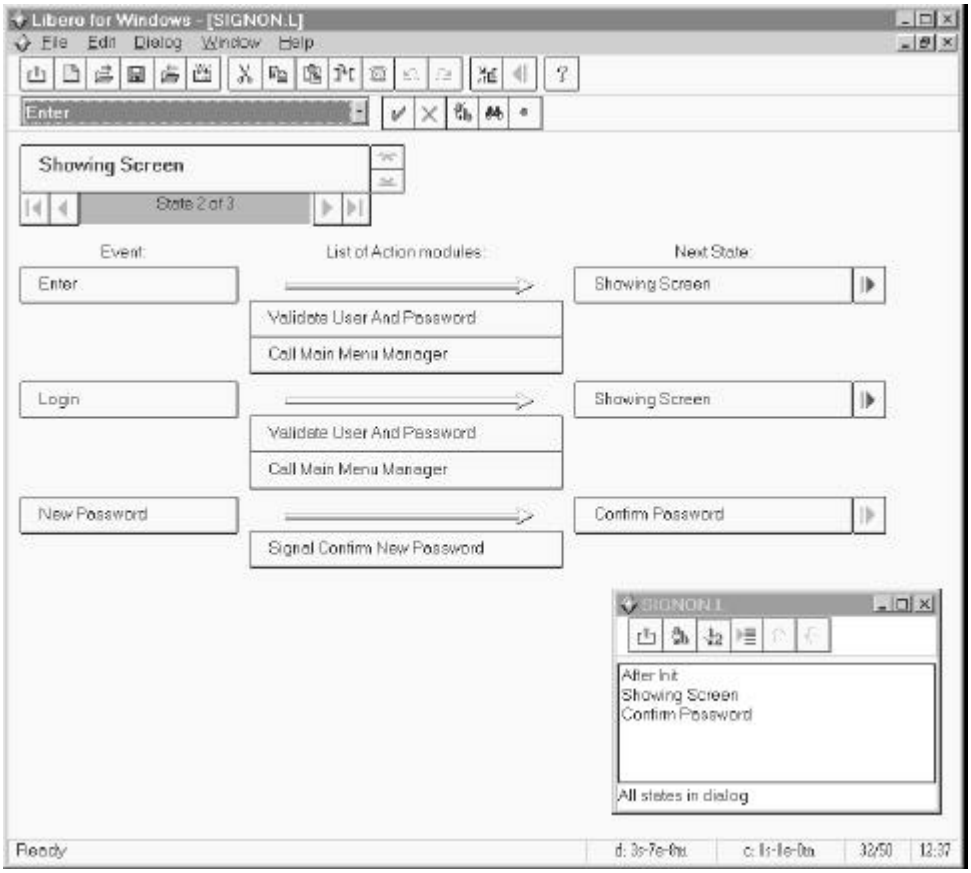
A Libero dialog formally describes a finite-state machine (although most people who write dialogs do not know this). Each state defines a set of valid events, actions for each event, and a next state, using this syntax:

```
State-Name:
    (--) Event-A                          -> Next-State
         + Action-Module
         + Action-Module...
    (--) Event-B                          -> Next-State
         + Action-Module...
    (--) Event-C                          -> Next-State
         + Action-Module...
```

Libero provides a Windows GUI for building dialogs, although the dialog can also be edited using a simple text editor. This is the dialog for a basic sign-on screen that accepts a user id and password, and also allows the user to enter a new password:

```
After-Init:
    (--) Ok                               -> Showing-Screen
         +
    (--) Error                            ->
         + Terminate-The-Program

Showing-Screen:
    (--) Enter                            -> Showing-Screen
         + Validate-User-And-Password
         + Call-Main-Menu-Manager
    (--) Login                            -> Showing-Screen
         + Validate-User-And-Password
         + Call-Main-Menu-Manager
    (--) New-Password                     -> Confirm-Password
         + Signal-Confirm-New-Password

Confirm-Password:
    (--) Enter                            -> Showing-Screen
         + Update-User-Password
    (--) Apply                            -> Showing-Screen
         + Update-User-Password
```

This is how the **Showing-Screen** state appears in the Libero GUI:



The program developer will provide code for these modules, which are implemented in C as functions:

```
validate_user_and_password
signal_confirm_new_password
call_main_menu_manager
update_user_password
terminate_the_program
```

The first time that the developer runs Libero, it generates a skeleton program and stubs for each of the modules:

```
$ lr signon

LIBERO v2.30 (c) 1991-97 iMatix <http://www.imatix.com>

lr I: processing 'SIGNON.L'...
lr I: creating skeleton program signon.c...
lr I: building signon.d...
lr I: building signon.i...
lr I: Building stub for validate_user_and_password
lr I: Building stub for call_main_menu_manager
lr I: Building stub for signal_confirm_new_password
lr I: Building stub for update_user_password
```

This is the skeleton program that Libero generates, edited for brevity. We highlight the modules that the programmer will complete:

```
/*=============================================================================*
 *                                                                             *
 *  signon.c    description...                                                 *
 *                                                                             *
 *  Written:    98/02/15    Your Name                                          *
 *  Revised:    98/02/15                                                       *
 *                                                                             *
 *=============================================================================*/

#include "signon.d"                     /*  Include dialog data             */
#include "signon.h"                     /*  Form definition file            */

/*- Global variables used in this source file only --------------------------*/

static SIGNON_DATA *form_data;          /*  Form data block                 */

/*********************************   M A I N   *********************************/

int signon_program (SESSION *p_session)
{
    session = p_session;                /*  Localise session block          */
    /*  Prepare to work with form                                          */
    /*  - - - - - - - - - - - - - - - - - Do nothing else here - - - - - - */
#   include "signon.i"                  /*  Include dialog interpreter      */
}

/************************   INITIALISE THE PROGRAM   ************************/

MODULE initialise_the_program (void)
{
    the_next_event = ok_event;
}

/************************   VALIDATE USER AND PASSWORD   ************************/

MODULE validate_user_and_password (void)
{
}

/************************   CALL MAIN MENU MANAGER   ************************/

MODULE call_main_menu_manager (void)
{
}

/************************   SIGNAL CONFIRM NEW PASSWORD   ************************/

MODULE signal_confirm_new_password (void)
{
}

/************************   UPDATE USER PASSWORD   ************************/

MODULE update_user_password (void)
{
}

/************************   TERMINATE THE PROGRAM   ************************/

MODULE terminate_the_program (void)
{
    the_next_event = terminate_event;
}
```

5.4.4 The Studio Broker System

iMatix Studio uses the WTP protocol as the basis for application organisation. The WTP protocol breaks an application into one or more *brokers*, each responsible for handling transactions to one or more application programs.

In this section we discuss the reasons for this organisation, and show how it affects the task of defining and managing an application.

5.4.4.1 Design Goals and Objectives

The broker system is intended to provide this functionality:

1. The ability to break an application into discrete components, each consisting of one or more programs. Each such component runs as a separate process. We call these '*application transaction processes*', or ATPs.

2. The ability to start an arbitrary number of each ATP, to provide a balance between the application load and the system load. Typically 10% of programs do 90% of the work, and are run so often that they must be present multiple times.

3. The ability to hide all this from the application developer. Such concerns are the problem of the application administrator.

5.4.4.2 Architecture Of The Broker System

The functions of a broker are specified by the WTP protocol. This figure shows how a broker works:

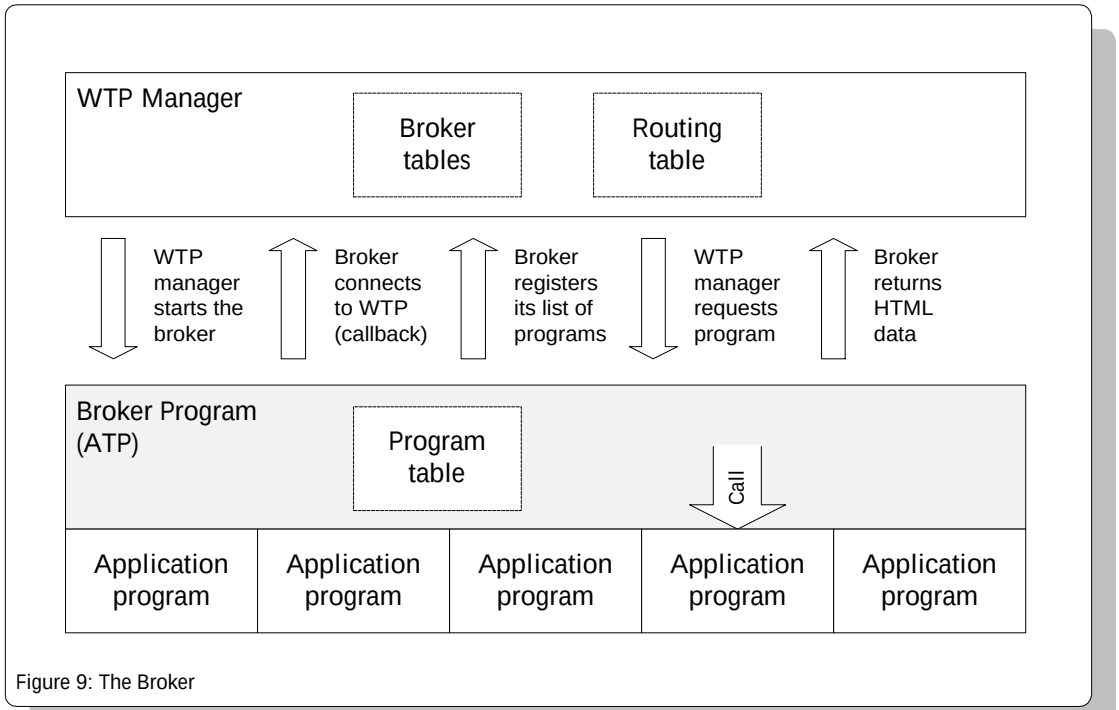


Figure 9: The Broker

A Studio broker program is always generated, although WTP brokers can be written by hand. The advantages of generating this code are that an otherwise complex and critical layer of code is fully hidden from the developer. Additionally, generating code allows us - for instance - to generate CGI brokers, so that an entire Studio application can run under CGI instead of WTP, transparently except for the loss of performance[5].

[5] Another limitation of Studio CGI applications is that all application programs must be linked into a single broker executable (ATP).

5.4.4.3 The Application Definition File

This figure shows how the application administrator manages the set of brokers for an application:

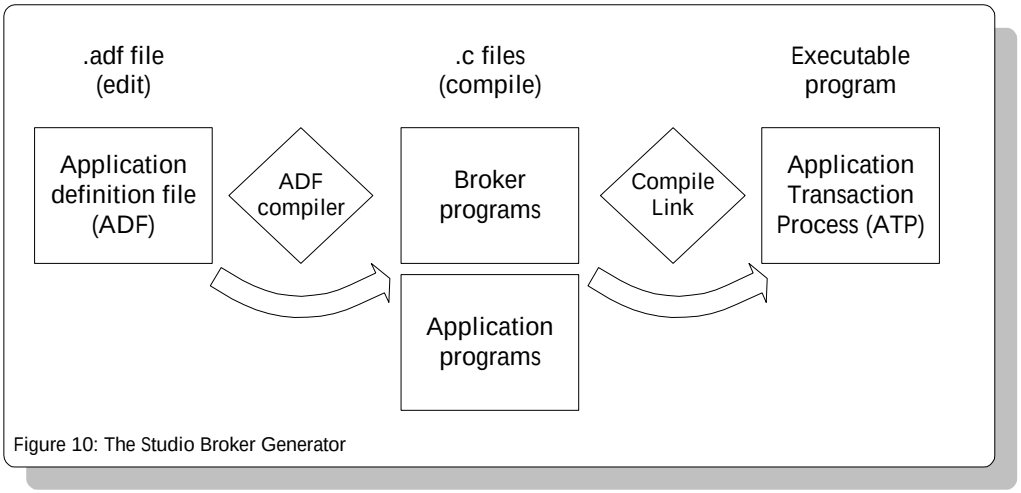


Figure 10: The Studio Broker Generator

Each independent application is defined by one ADF. This is a typical ADF for a toy application:

```
#
#   example.adf - Example of a toy ADF
#

Application = {
    Title = "Example application"
    Name  = example
    Model = wtp
    Root  = signon

    server = {
        Name = atp001
        Type = c
        program = { name = signon   }
        program = { name = topmenu  }
        program = { name = bugtlist }
    }
    server = {
        Name = atp002
        Type = c
        program = { name = budget   }
        program = { name = supplr   }
        program = { name = pchsord  }
    }
}
```

When one runs the ADF compiler, it regenerates the various broker programs:

```
$ adfgen example

adfgen - Studio ADF compiler V2.0
Copyright (c) 1996-98 iMatix - http://www.imatix.com

adfgen I: processing example.adf...
```

By generating the broker programs, and thus fixing the broker program tables at compile time, we avoid a number of problems:

1. No extra configuration files are needed.

2. The link step can automatically bind all the programs required for an ATP server, since these are referenced correctly in the broker.

This is a typical generated broker program, edited for brevity. Note that as usual with Studio, the generated code is well commented and readable:

```
/*===========================================================================
 *
 *  wkbench - Studio Workbench broker
 *  Generated by adfgen from brschwtp.c on 1998/02/20, 22:04:25.
 *
 *===========================================================================*/

#include "sfl.h"                        /*  SFL prototypes & definitions     */
#include "wtplib.h"                     /*  WTP definitions                  */
#include "wtpmsg.h"                     /*  WTP message API                  */
#include "formio.h"                     /*  Formio definitions               */
#include "browtp.h"                     /*  WTP broker definitions           */

static void handle_signal (int the_signal);

int main (int argc, char *argv [])
{
    char
        *version,                       /*  WTP version string               */
        *protocol,                      /*  Protocol to use                  */
        *port,                          /*  Port number                      */
        *callback;                      /*  Callback key                     */

    if (argc < 4)
      {
        puts ("Must be run from WTP manager!");
        exit (0);
      }
    else
      {
        version  = argv [1];
        protocol = argv [2];
        port     = argv [3];
        callback = argv [4];
      }
    wtp_open     (version, protocol, port);
    wtp_connect  (callback, argv [0]);
    wtp_register ("signon",  1);
    wtp_register ("topmenu", 0);
    wtp_register ("budget",  0);
    wtp_ready    ();

    /*  Pass to Transaction Manager                                          */
    wtp_broker ();

    return (EXIT_SUCCESS);
}

/*  We must define each program that the broker contains, with a C
 *  prototype and an entry in the broker map table.
 */

int signon_program  (SESSION *session);
int topmenu_program (SESSION *session);
int budget_program  (SESSION *session);

BROKER_MAP wtp_broker_map [] = {
    { "signon",  signon_program  },
    { "topmenu", topmenu_program },
    { "budget",  budget_program  },
    { NULL, NULL }
};
```

5.4.5 Other Technical Issues

5.4.5.1 Context Management

The problem of *context management* reappears on all transaction-processing systems. The problem arises because the same executable program is shared by many sessions, one after the other. Therefore any data in the program's memory is in an undefined state when the programs starts to process a transaction for some session.

There are many ways of resolving this. After working on systems such as IBM MVS/CICS, Digital VAX/ACMS, and Bull TDS, we have gained considerable experience with the problem and its possible solutions. Generally-speaking, the possible solutions are:

1. To write stateless, context-free programs. This is essentially how the web CGI (common gateway interface) protocol works. It is a very poor approach for business application programs, since state and context are essential for efficient and rich processing.

2. To pass a minimum of context in the client-to-server message. This is a hybrid technique, used by some web CGI programs that absolutely require context. The context can be information such as a database key, a user name and password, etc. This approach is useful so long as no real context is passed, but a *reference* to some context. Passing real context is both slow (since large amounts of data may have to be passed across a slow network) and insecure (since internal program data is passed across the network).

3. To store all application context in a single session block, passed between all application programs. This is the standard technique used by the above transaction-processing systems. It can be efficient, but is constricting when building orthogonal systems: one change in one program can potentially affect thousands of programs.

4. To simulate a normal single-user program by automatically saving and restoring context *behind-the-scenes*. This is possible for programs written in languages like COBOL, where all data is statically allocated in a contiguous area. It is difficult for C programs, which can do arbitrary memory allocation.

5. To provide the application programmer with specific areas that will be saved and restored automatically, on a per-program basis. This combines the principles of (3) and (4), and appears to be the best solution for C programs.

Efficient context-management can also use a data-compression algorithm to reduce the amount of memory used, and a buffering algorithm to write contexts to disk when memory space is limited.

iMatix Studio implements context management using the last technique listed above. Context management is supported by the WTP protocol.

5.4.5.2 Application Security

The question of security comes up early in any discussion about Internet and intranet applications. We believe that a solid approach to security is necessary, but that this is not something that must be built into the technical layers. Any Internet connection exposes the server systems to a range of security issues, through protocols such as FTP, telnet, SMTP, etc. These protocols are beyond the control of the iMatix Studio layer, and must be

handled by the appropriate use of *firewalls*, TCP/IP loggers, analysers, etc. This issue covers *all* Internet applications including e-mail, file-transfer, remote log-ins, and iMatix Studio.

Experience tells us that the best point to place user-identification control is at the entry point to the application, in a *log-in screen*. This ensures that the user is authorised *for that application* and that appropriate session parameters such as authorisation level, language, etc. for that application are used in all programs.

Application security should not be confused with data security, which allows sensitive data to be passed across unsecured lines. iMatix is working on SSL/3 (secure socket layer, version 3) and TLS (an evolution of SSL/3) support for Studio; these protocols allow data to be encrypted automatically between the server and the browser.

5.5 Still Under Development

These functions and tools are currently under development:

1. Secure socket support (SSL/3 and TLS).

2. Support for C++, server-side Java, Perl, and COBOL.

3. Dictionary support for database I/O code generators.

4. Database schemas for Informix, Ingres, DB2, SQL Server. Schemas for Oracle and ODBC already exist.

5. The Studio Workbench application, which packages the various command-line tools into a project-based front-end.

6. Repacking of the iMatix WTP manager as plugins for IIS, Netscape, and Apache servers.

6. Problems Solved by iMatix Studio

6.1 Technical Problems Solved

1. The problem of building applications that can scale up to handle thousands of users - iMatix Studio was designed from the start to handle the heaviest loads.

2. The problem of deploying applications on different systems in a network - iMatix Studio produces fully-portable applications.

3. The problem of writing for clients systems that are beyond your control - iMatix Studio uses the common Web browser as its client.

4. The problem of working with technologies that keep changing - iMatix Studio works with tried and tested technologies, with your choice of programming language, and your choice of database.

5. The problem of working with technologies that take six months to master - iMatix Studio can be used within a few days of starting, and can be mastered in two weeks.

6. The problem of working with technologies that are opaque and hard to understand - iMatix Studio is based on open technologies, such as SFL, that have been tried and tested by thousands of users for several years.

6.2 Non-Technical Problems Solved

1. The problem of developing effective web applications - iMatix Studio lets you design applications with rich, multi-page interfaces.

2. The problem of hiring highly-skilled people - iMatix Studio is designed for people with good knowledge of the application, but only modest technical skills. When we develop a payroll package, we want payroll experts, not Java gurus.

3. The problem of maintaining quality in the application - by using Libero and the form I/O code generators, the essential core of every program is kept as readable and accurate high-level designs.

4. The problem of developing a commercial prototype - the form I/O system lets you quickly develop prototype forms and programs that can be used to sell and launch a project. A Studio application can be put onto a single diskette, and will run without an installation procedure.

7. Conclusions

It is our experience that the cost of a technical platform is outweighed many times by the costs of the functional solution. These costs are:

1. Cost of people.

2. Efficiency of the design and development process.

3. Efficiency of the resulting application programs under realistic conditions.

4. Cost of the client platform.

The technical platform is a make-or-break issue for any application development. A major cause of project overruns or failures is the inability to control a complex and unstable technical platform.

After looking at the alternatives, and studying the experience of teams that have worked with iMatix Studio, we conclude that:

- The alternative technologies for application development are either limited to non-web approaches, proprietary, limited to small applications, or/and too complex for daily use.

- iMatix Studio offers a real alternative based on tried and tested techniques, bringing a decade of successful application development principles to the Internet and intranet.

- iMatix Studio can be used by non-technical developers with a minimum of training.

- Studio applications run quickly, with little difficulty, and with minimum manual intervention.

- Studio applications are cheap to develop, cheap to deploy, and cheap to maintain.

- This translates directly into real gains, financially and in project schedules.

8. Appendix A - The Pyramid Principle

8.1 The Pyramid Principle

iMatix Studio uses the *Pyramid Principle* as a method for simplification and abstraction. This model is based on the generally-observed phenomenon that 90% of any system will get 10% of the resources, and 10% will get the remaining 90%. This applies to software development as well as to national economies.

The main application of the Pyramid Principle is to divide a software application into the 10% and 90% parts, then work accordingly. This figure shows the division:

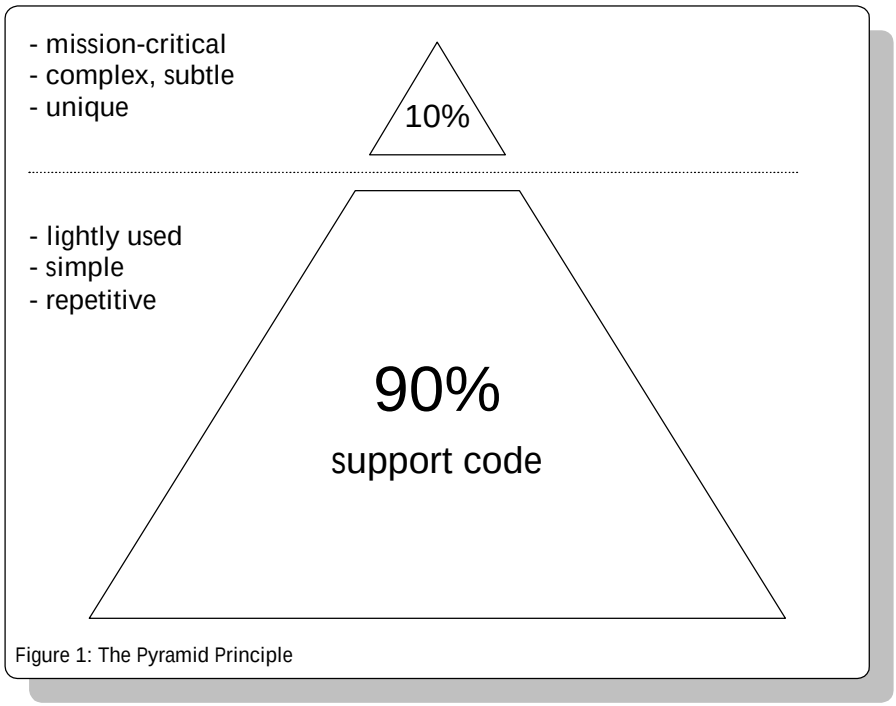


Figure 1: The Pyramid Principle

The ratios may be different (5%-95% or 20%-80%) but in our experience, this division is always possible, and always desirable. By defining this principle from the outset, we can reduce costs and ensure a better result. Many projects pay heavily for treating support code as mission-critical, or vice-versa.

When we apply the Pyramid Principle to application development, we see that:

- In any application, the mission critical programs or 'screens' are a small part of the total source code, but account for a majority of the development effort.

- These programs also account for a majority of system resources (CPU time), bug reports, user time, etc.

- These programs are typically very complex, functionally, and require developers who are both experienced and competent.

- It is therefore pointless to try to generate such programs, to simplify them, or to develop them with less skilled developers.

By contrast, when we look at the support code we see that:

- The support programs account for a majority of the total code.

- The support programs are not significant in terms of system usage, largely because the application users invoke these programs very infrequently.

- These programs are typically very simple, functionally, and can be written by less-skilled developers, or generated from templates (naturally such templates *do* require the input of a lot of skill).

- It is therefore pointless to write such programs by hand (unless this process is guided by strict templates), or to optimise[6] them.

We know that quality and performance are economic decisions. The Pyramid Principle lets us direct the effort at those points where it provides the best value-for-money.

[6] It is well-accepted in software development that 'optimisation' is only positive when applied to 'hot-spots' in an application. In all other places, well-written, but *not necessarily optimal* code is most reliable, cost-effective, and appropriate.

9. Appendix B - Software Portability

9.1 Software Portability

This article was written by Pieter Hintjens of iMatix Corporation for Dr Dobb's Journal *in 1997, and is reprinted with kind permission.*

Software portability is a key part of the business of producing good, long-lasting software on time and cheaply. Since this is our business, we needed a tool for making portable applications for as wide a range of systems as possible. In this article, we describe how we built our freely available solution for C applications, the Standard Function Library. We'll examine our benchmark application, the Xitami web server (see DDJ Special Report on Software Careers, Spring 1997), and the way it runs on UNIX, Windows, and OS/2.

9.1.1 Portability Defined

We define 'portability' as a set of concrete goals:

- the application must run on any supported system (e.g. for a web server, any operating system with an ANSI C compiler and TCP/IP support) without modification;
- the application must contain no non-portable code (i.e. no specific logic for system peculiarities);
- the programmer must be able to work without system-specific knowledge (e.g. how to create a child process on such-and-such a system).

A portable approach is not always possible. When it is possible, however, the advantages of a portable approach over non-portable, native development are overwhelming:

- the market for the application is much larger;
- changes in operating systems are absorbed cheaply and quickly;
- you need less specialised knowledge;
- portable code will survive platform changes, and so is useful for longer;
- portable code is cheaper to develop;
- portable code is more robust;
- a portable approach can be faster and give better results than a non-portable, native, approach.

This last statement is counter-intuitive, and needs justification. We'll show how this worked for the Windows version of our Xitami web server.

Our basic design divides each application into two layers: a "technical layer" and a "functional layer." The technical layer, reused in all applications, encapsulates all non-portable features, and provides a set of useful library functions. The functional layer is portable, and constitutes the real "application."

Our technical layer is a library of C functions that we called the ***Standard Function Library*** (SFL). SFL is further subdivided into different packages. For instance, the string package provides various string manipulation functions, the socket I/O package provides a set of functions to access Internet sockets, the date package provides a set of date conversion functions, and so on.

9.1.2 Abstracting Functions

The SFL provides a series of abstractions for functions that we need on all systems, but that must be implemented with non-portable, system-specific code. It also encapsulates functionality, as does any useful library. We generally combine these two needs.

For example, the process control package, sflproc.c, provides functions to create, monitor, and kill background processes. Under UNIX we create a child process by using the fork() and exec() system calls. Under 32-bit Windows we use the CreateProcess() system call. But the implementation is not trivial: we need quite a lot of supporting code to redirect input and output streams, to ensure that the child process is correctly started, and to handle errors.

Our abstraction for process_create() looks like this:

```
PROCESS                 /*  Returns a PROCESS token          */
process_create (
    char *filename,     /*  Name of file to execute          */
    char *argv [],      /*  Arguments for process, or NULL   */
    char *workdir,      /*  Working directory, or NULL       */
    char *std_in,       /*  Stdin device, or NULL            */
    char *std_out,      /*  Stdout device, or NULL           */
    char *std_err,      /*  Stderr device, or NULL           */
    char *envv [],      /*  Environment variables, or NULL   */
    Bool  wait          /*  Wait for process to end          */
)
```

If you abstract functions correctly, you don't lose performance or functionality. Of course, the "correct" way is not always obvious, and sometimes takes a few iterations to discover. Often, when faced with various options, we choose something closest to the UNIX model. In the last few years, many platforms are moving towards Posix compatibility. One consequence of this is that the Posix, UNIX-like abstraction is the most stable and long-term. The test of a good portability abstraction is that it does not change when it's ported to a new platform. Hindsight, experience, and access to a lot of documentation helps a lot here.

One well-understood benefit of packaged functionality is that the internals - what we call the technical layer - can be improved as needed without affecting the calling programs. For an SFL package, this "improvement" can mean porting to a new platform, or improving the way it works on a specific platform. When Ewen McNeill <ewen@imatix.com> ported the SFL to OS/2, he relied on the fact that OS/2 with EMX provides many UNIX-like functions. So, it was quite easy to make the process control package work on OS/2. Native OS/2 system calls provide better performance, but we're not obliged to make an optimal solution right away.

9.1.3 Case Study - a Portable Web Server

A project like the SFL needs several real, demanding client applications to test the technology and prove that it works. We decided to use the SFL in all our software development, starting with our web server, Xitami. In an Internet server program -- which has little or no user-interface -- the main portability issues are file handling, process control, and socket I/O.

The SFL file-handling package sflfile.c hides the differences between UNIX, VMS, and MS-DOS text file formats and filename syntaxes. For instance, the file_where() function searches along a path for some file. This is a simple concept, very useful to locate an applications' data files, but works quite differently on VMS, UNIX, or MS-DOS. The file_is_executable() function checks the file's attributes under UNIX, but actively searches for an .exe, .com, or .bat file under MS-DOS or OS/2. There are many issues that we do not cover (e.g. file locking); however such package such as the sflfile.c provides a framework for adding new functions as we need them.

The socket I/O package, sflsock.c shows how to avoid portability problems by a combination of simplification and stubborn audacity. The socket abstraction is itself widespread and pretty standard, being based on the BSD socket library. A simple socket-based program is easily portable to all BSD-derived systems: UNIX, OS/2, and Digital OpenVMS. When we looked at using socket-based programs under Windows, however, we found two problems.

Firstly, the Windows socket library (Winsock) has a call interface that only partially resembles the BSD socket interface. Secondly, under Windows, sockets are usually connected to the user-interface code -- socket events being handled in the same way as mouse and keyboard events. This means that a Windows socket program is constructed totally differently from a BSD-style socket program.

Our preferred abstraction for sockets (as for anything else) is a set of simplified functions (connect, read, write,...) that both encapsulate the differences between systems, and add scaffolding code such as error handling. The question was: would such an abstraction work under Windows? Winsock provides the select() system call, which lets us collect socket events in Windows as we do in UNIX. So we basically cheat, and answer the portability question by avoiding Window's normal event-driven socket handling altogether. Surprisingly, perhaps, this works just as efficiently as the 'native' Windows approach.

The SFL does not address portability in user-interfaces. We're basically interested in programs that do little or no user input/output. When we compiled the web server, using Microsoft Visual C/C++ 4.0, as a console program, it worked much like its UNIX counterpart -- that is, simply and quickly. However, we wanted to build a Windows user-interface for the web server, mainly for marketing reasons. A common criticism of a 'portable' approach is that it restricts one to producing simplistic or functionally-poor applications. The Windows version of Xitami proves that this is not true; indeed, that a portable approach can give better results, and faster, than the system-specific approach.

We wrote a Windows user-interface program as two threads. The first thread manages a user-interface dialogue, and is not portable (we wrote different versions for 16-bit and 32-bit Windows, and for the Windows NT service version). The second thread runs like a classic UNIX daemon, in the background. The two threads communicate via shared variables.

It took about a week to build this front-end. The result is application that looks and feels like a native Windows application, but where the bulk of the application code is portable, and kept cleanly separate from the non-portable user-interface code.

To demonstrate the effectiveness of this approach: Xitami now runs on Windows 3.x as a 16-bit program, on Windows 95 and NT as a 32-bit program, on Windows NT as a service, and on Windows 95 and NT as a 32-bit DOS program. In all cases the bulk of the code remains unchanged, and well-isolated from the thin UI layer required to glue it to the specific Windows version. In contrast, native Windows web servers are highly version-specific. For instance, Microsoft's own IIS runs only on NT 4.0. Yet, Xitami is faster than IIS, and a relatively small program (the complete executable code is about 400 Kbytes).

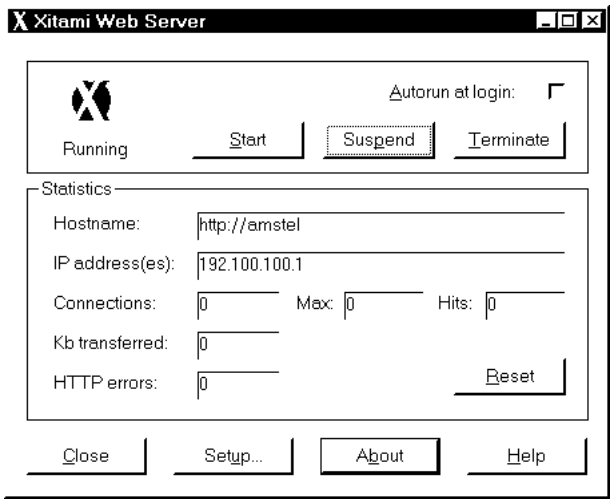


Figure 1: The Xitami Control Panel

9.1.4 Portability and the C Language

So far, we've discussed techniques that can apply to any generally-portable programming language -- many of the principles were developed by Leif Svalgaard <leif@ibm.net> and others in the 1980's for supporting portable COBOL applications. The C language presents its own peculiar challenges; mainly non-portable header files and library functions, and non-portable language constructs and data types.

Let's look at header files. The principle is that you add an `#include` statement for the various functions that you need. The problem is that on different platforms, these files (except for standard ANSI header files) sit in different directories, have different names, or may not even correspond at all.

For example, on most UNIX systems to use socket functions you must include the files <sys/socket.h>, <netinet/in.h>, and <arpa/inet.h>. Under IBM AIX you also need <sys/select.h>. Under Digital VMS you need only <socket.h> and <in.h>. Under Windows, you need <winsock.h>, and under OS/2 with EMX, you need <sys/socket.h>, <sys/select.h>, <netinet/in.h>, and <arpa/inet.h>.

At some point, this requires conditional macros (e.g. "#if __TURBOC__"). The question for us was how to do this with the least effort. Listing one shows how a careful programmer might include the socket header files in a program.

Different compilers predefine macros like WIN32, __i386__, __vax__, or _hpux. In each source program that uses socket functions or socket data types, our careful programmer could repeat these conditional macros. Eventually our programmer would see that copying code like this was a bad idea, and might create a single header file to handle the various includes. Of course, the same scenario applies to other system functions - files, directories, processes,... So, with some more work, our programmer could build a set of header files, each covering some set of functions.

This fictitious set of header files now poses its own maintenance burden. One day our programmer decides to compile the code on a new platform, or using a new release of the C compiler. There are compile errors - each of the header files has to be changed to take into account new predefined macros, header file locations, and other changes.

Looking at this scenario, we saw a lot of unnecessary hard work. Our solution was two-fold. To start with, we decided never to use compiler-specific macros like __i386__. Instead, we would define a stable, clean set of macros (__UNIX__, __VMS__, __OS2__, __MSDOS__,...) using whatever internal mechanisms necessary. Our careful programmer can write conditional code ("#if __OS2__"), but when we want to handle a new compiler, there is at most one file to change. Secondly, we would bite the bullet and put all #include statements in one place. Again, the aim was to have a single point of focus for non-portable header files.

We wrote a single header file, prelude.h, that handles both these needs. Listing Two shows the way that system-specific header files are included in prelude.h. This is a controversial approach: we had some heated discussions in comp.lang.c about its (de)merits. The decision about exactly which header files to include is not evident. (We take most, but not all, ANSI header files. Then we take those files needed to support sockets, directory access, timers, etc.) A program that uses the prelude.h will compile slower (two to five times slower) than a program that includes only those files it really needs. The prelude.h file makes a raft of definitions that may conflict with those the programmer wants to make. The final complaint is that we remove a level of control that many C programmers are used to exercising.

However, we consider this as an inevitable and worthwhile compromise along the way to true portability, as well as a good way to simplify an otherwise complex problem. We've used the prelude.h file in many projects, with or without the SFL, and it works well.

Another portability gotcha in C is the size of integer datatypes. On most systems an 'int' is 32 bits, but on some it's 16, and on others 64. Here we took a commonly-used approach, which is to define datatypes 'byte', 'dbyte', and 'qbyte'. These are always one byte, two bytes, and four bytes long. This approach probably rules-out systems with exotic words sizes, a fair compromise for us. Again, we put these type definitions into prelude.h, so that all programs would share them.

A final problem with writing portable C code is that the programmer must be careful to avoid non-portable library functions and data types. This is a matter of experience, good books, and a good compiler help function. Most compiler help systems will indicate whether a function is ANSI, POSIX, or system-specific. We assumed that all ANSI functions are supported on our target systems. We then rewrote common but non-standard functions such as strlwr(). We also assumed that most platforms will move towards POSIX compatibility, so we use POSIX constructs where possible. Largely, this approach works, though we find that it is important to regularly 're-port' applications to target platforms to ensure that non-portable constructs do not creep in.

Portable code built this way can be simple but functional. Listing Three shows a portable directory listing program. This is a typical example of functionality ('directories') that exists on many platforms, which is useful in applications, but which requires non-portable code to use.

9.1.5 Redefining The Makefile

When we moved our C programs to various platforms, we found that compilers are not standard, not even on different versions of the same operating system. The C compiler is generally called 'cc', but often uses different arguments for optimisation, enforcing ANSI syntax, linking, etc.

The approach used in many multiplatform (but not portable) products is to build these differences into many makefiles, one per platform. Alternatively, to build the differences into a single multiplatform makefile. In any case, the user must choose a target system by issuing a command like 'make aix'.

To rebuild a package like the SFL, we have to issue a number of compile commands, create an archive file for the compiled programs, and link some executable programs. We decided that writing and maintaining traditional UNIX makefiles was too much work, and only a partial solution. There are tools that generate platform-specific makefiles: for example the **imake** tool used in the X window system. **imake** is powerful, but too complex for our needs, and too specific to UNIX platforms. Our needs were simple: recompile a set of C programs, build some object libraries, link some executables.

Taking a cue from the CERN libwww reference library installation, we wrote a 'c' script that detects the UNIX platform, and runs the appropriate commands to compile or link a program. The 'c' script fully abstracts the compiler interface on UNIX platforms. The result of this is that the programmer is more portable, as well as the software.

We additionally wrote a small tool that generates build scripts from a single portable command file. The tool (**otto**) generates scripts for UNIX (using the 'c' script), VMS, OS/2, and several MS-DOS compilers, and is easy to extend for other platforms. An **otto** script can test for required files, compile and link programs, copy, rename, and delete files, and run system-specific commands. To make life easier for the developer, **otto** also develops standard makefiles.

9.1.6 C and UNIX Portability Standards

In 1983, the ANSI C X3J11 Committee started the process of defining a standard C language that would run on all platforms, from embedded micro-controllers to supercomputers. ANSI C was published in 1990 as ANSI X3.159-1989, and was later replaced by the ISO standard ISO/IEC 9899:1990. The most significant difference between the earlier de-facto standard, K&R C, and ANSI C was a standard C run-time library with corresponding header files and function prototypes. Most (but not all) modern C compilers support ANSI C fully. One notable exception is SunOS, where the default compiler is K&R, and an ANSI C compiler must be separately purchased. A new ANSI standard is expected in 1999.

System V.4 UNIX is (or rather, was, until AT&T sold the UNIX trademark) the de-facto "standard" commercial UNIX, and provides "portability by inclusion": it combines various UNIX offshoots: System V.3.2, 4.3BSD, SunOS, XENIX. IBM's AIX, HP's HP/UX, and Sun's

SunOS are modified versions of System V.4 or the earlier System V.3. After many years of industry conflict over the "UNIX" trademark and copyrights, Novell received the UNIX trademark from AT&T, and then passed it to X/Open, who started to define a "standard UNIX".

OSF/1 is a consortium standard for UNIX from the Open Systems Foundation, started by a few large vendors as reaction to the UNIX policies adopted by AT&T. Digital's UNIX systems now use OSF/1, and IBM has stated its intention to replace their AIX operating system (largely based on 4.3BSD) by OSF/1. OSF/1 is principally System V.4 compatible, and also tries to support POSIX.1. Like System V.4 it often provides two or more alternative versions of system functions.

In 1995, X/Open and OSF were brought together under the Open Group, finally providing the basis for a single, standard UNIX.

POSIX -- unlike the commercial UNIX variations -- is an official standard from the IEEE. POSIX is actually a set of standards covering operating systems, programming languages, and tools. The POSIX.1 standard (IEEE 1003.1-1990) covers operating systems and looks much like a subset of UNIX. Non-UNIX operating systems (such as Digital's OpenVMS, Windows NT, and even IBM MVS) can and probably will eventually be POSIX compliant.

The promise is this: use only POSIX functions and your applications will be portable to all POSIX-compliant operating systems. In reality, POSIX.1 standardises a number of important system interfaces, but is not rich enough to provide the basis for many real-life applications. In an example of remarkably bad marketing, the IEEE does not make its standards documents freely available; these must be bought. The web site http://www.posix.com/ provides more information on obtaining IEEE standards documents.

9.1.7 Conclusions

The test of a portability toolkit is moving to a new system. In January 1997, Ewen McNeill <ewen@imatix.com> ported the SFL to OS/2. Ewen made changes for OS/2 to four SFL packages: the user-ID package, the socket package, the directory package, and the process-control package. Ewen also added OS/2 support to the 'c' script and Otto. In February 1997, Vance Shipley <vances@motivity.ca>, ported the SFL to SCO UNIXWare and SCO OpenServer 5.0, changing the 'c' script, the prelude.h file, and the process-control package. Again, the changes were minor, quickly made and tested. In both cases, as we expected, all our applications built on the SFL - including Xitami - ran without modification. Well, not exactly, because OS/2 showed-up a couple of dormant bugs.

Portability need not be a constraint on the development process. Rather, it is a complexity-reduction method that can help the developer deliver high-quality, stable, and efficient applications at a lower cost. Portability applies to software, to processes, and to people.

We built our portability toolkit by:

1. identifying the types of application we wanted to support;

2. identifying the types of target systems we wanted to support;

3. finding an abstraction (an API) to protect the programmer from the differences in these systems;

4. writing a function library to support this abstraction;

5. writing the tools to support this abstraction.

9.1.8 For Further Reading

"Internetworking With TCP/IP Volume III: Client-Server Programming And Applications BSD Socket Version" by Douglas E. Comer and David L. Stevens, published 1993 by Prentice-Hall Inc. ISBN 0-13-020272-X. This is the bible on writing internet servers.

"Porting UNIX Software", by Greg Lehey, published 1995 by O'Reilly & Associates, Inc. ISBN 1-56592-126-7. An excellent book that helped us avoid most of the known UNIX portability pitfalls.

"Software Portability with imake", by Paul DuBois, published 1993 by O'Reilly & Associates, Inc. ISBN 1-56592-055-4. Provides a good description of the imake tool and its possibilities.

9.1.9 Source Listings

9.1.9.1 Listing one: how *not* to use socket header files

```
/*  Listing one: how *NOT* to use socket header files                       */

/*  Include files for Windows                                               */
#if defined WIN32 || defined _WIN32 || defined WINDOWS || defined _WINDOWS
#   include <windows.h>
#   include <winsock.h>

/*  Include files for OS/2                                                  */
#elif defined __EMX__ && defined __i386__
#   include <sys/socket.h>
#   include <sys/select.h>
#   include <sys/time.h>
#   include <sys/stat.h>
#   include <sys/ioctl.h>
#   include <sys/file.h>
#   include <sys/wait.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>

/*  Include files for Digital OpenVMS                                       */
#elif defined VMS || defined __VMS || defined __vax__
#   include <socket.h>
#   include <in.h>

/*  Include files for UNIX, except AIX                                      */
#elif defined unix || defined __unix__ || defined __hpux || defined SUN
#   include <sys/socket.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>

/*  Include files for IBM AIX                                               */
#elif defined _AIX || defined AIX
#   include <sys/socket.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>
#   include <sys/select.h>

#endif
```

9.1.9.2 Listing two: an extract from the Universal Header File

```
/*  Listing two: an extract from the Universal Header File                  */

#if (defined WIN32 || defined (_WIN32))
#   undef __WINDOWS__
#   define __WINDOWS__
#   undef __MSDOS__
```

```
#   define __MSDOS__
#endif

/*  __OS2__     Triggered by __EMX__ define and __i386__ define to avoid    */
/*              manual definition (eg, makefile) even though __EMX__ and    */
/*              __i386__ can be used on a MSDOS machine as well.  Here      */
/*              the same work is required at present.                       */
#if (defined (__EMX__) && defined (__i386__))
#   undef __OS2__
#   define __OS2__
#endif

#if (defined (__hpux))
#   define __UTYPE_HPUX
#   define __UNIX__
#   define _INCLUDE_HPUX_SOURCE
#   define _INCLUDE_XOPEN_SOURCE
#   define _INCLUDE_POSIX_SOURCE
#elif (defined (_AIX) || defined (AIX))
#   define __UTYPE_IBMAIX
#   define __UNIX__
#elif (defined (linux))
#   define __UTYPE_LINUX
#   define __UNIX__
# … etc
#elif (defined __UNIX__)
#   define __UTYPE_GENERIC
#endif


/*- Standard ANSI include files ---------------------------------------------*/

#ifdef __cplusplus                      /*  PA 96/05/29                     */
#include <iostream.h>                   /*  A bit of support for C++        */
#endif

#include <ctype.h>
#include <limits.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <time.h>
#include <errno.h>
#include <float.h>
#include <math.h>
#include <signal.h>
#include <setjmp.h>


/*- System-specific include files -------------------------------------------*/

#if (defined (__MSDOS__))
#   if (defined (__WINDOWS__))          /*  When __WINDOWS__ is defined,    */
#       include <windows.h>            /*    so is __MSDOS__               */
#       include <winsock.h>            /*  May cause trouble on VC 1.x     */
#   endif
#   if (defined (__TURBOC__))
#       include <dir.h>
#   endif
#   include <dos.h>
#   include <io.h>
#   include <fcntl.h>
#   include <malloc.h>
#   include <sys\types.h>
#   include <sys\stat.h>
#endif

#if (defined (__UNIX__))
#   include <fcntl.h>
#   include <netdb.h>
#   include <unistd.h>
#   include <dirent.h>
#   include <pwd.h>
#   include <grp.h>
#   include <sys/types.h>
#   include <sys/param.h>
#   include <sys/socket.h>
#   include <sys/time.h>
#   include <sys/stat.h>
#   include <sys/ioctl.h>
#   include <sys/file.h>
#   include <sys/wait.h>
#   include <netinet/in.h>
```

```
#   include <arpa/inet.h>
/*  Specific #include's for UNIX varieties                                  */
#   if (defined (__UTYPE_IBMAIX))
#       include <sys/select.h>
#   endif
#endif

#if (defined (__VMS__))
#   if (!defined (vaxc))
#       include <fcntl.h>              /*  Not provided by Vax C           */
#   endif
#   include <netdb.h>
#   include <unixio.h>
#   include <types.h>
#   include <socket.h>
#   include <dirent.h>
#   include <time.h>
#   include <pwd.h>
#   include <stat.h>
#   include <in.h>
#endif

#if (defined (__OS2__))
/*  Include list for OS/2 updated by EDM 96/12/31
 *  NOTE: sys/types.h must go near the top of the list because some of the
 *        definitions in other include files rely on types defined there.
 */
#   include <sys/types.h>
#   include <fcntl.h>
#   include <malloc.h>
#   include <netdb.h>
#   include <unistd.h>
#   include <dirent.h>
#   include <pwd.h>
#   include <grp.h>
#   include <sys/param.h>
#   include <sys/socket.h>
#   include <sys/select.h>
#   include <sys/time.h>
#   include <sys/stat.h>
#   include <sys/ioctl.h>
#   include <sys/file.h>
#   include <sys/wait.h>
#   include <netinet/in.h>
#   include <arpa/inet.h>
#endif
```

9.1.9.3 Listing three: directory list program

```
/*
 *  Name:       testdir.c
 *  Title:      Test program for directory functions
 *  Package:    Standard Function Library (SFL)
 *
 *  Written:    96/04/02 <sfl@imatix.com>
 *  Revised:    96/12/12 <sfl@imatix.com>
 *
 *  Synopsis:   Testdir runs the specified or current directory through
 *              the open_dir and read_dir functions, formatting the output
 *              using format_dir.
 *
 *  Copyright:  Copyright (c) 1991-1998 iMatix Corporation
 *  License:    This is free software; you can redistribute it and/or modify
 *              it under the terms of the SFL License Agreement as provided
 *              in the file LICENSE.TXT.  This software is distributed in
 *              the hope that it will be useful, but without any warranty.
 */

#include "sfl.h"

void handle_signal (int the_signal)
{
    exit (EXIT_FAILURE);
}

int main (int argc, char *argv [])
{
    NODE    *file_list;
    FILEINFO *file_info;
    char     *sort_type = NULL;

    signal (SIGINT,  handle_signal);
    signal (SIGSEGV, handle_signal);
    signal (SIGTERM, handle_signal);
```

```
    if (argc > 2)
        sort_type = argv[2];

    file_list = load_dir_list (argv [1], sort_type);
    if (file_list)
      {
        for (file_info  = file_list-> next;
             file_info != (FILEINFO *) file_list;
             file_info  = file_info-> next
            )
            puts (format_dir (&file_info-> dir, TRUE));
        free_dir_list (file_list);
      }
    /*  Check that all allocated memory was released  */
    mem_assert ();
    return (EXIT_SUCCESS);
}
```

10. Appendix C - The Web Transaction Protocol

10.1 Introduction

This document describes the Web Transaction Protocol version 1.0 (WTP/1.0), a protocol that adds transaction processing functionality to HTTP ('web') servers. WTP is an open protocol that can be implemented in many ways. This document defines a reference implementation, which corresponds to the WTP implementation offered in the iMatix Web Transaction Server.

10.1.1 Overview

The Web Transaction Protocol (WTP) is a replacement for the common gateway interface (CGI) protocol commonly used to build web application programs. WTP corrects some well-known deficiencies in the CGI protocol, and adds transaction management functions specifically required by large-scale applications.

The main difference between CGI and WTP is that CGI is designed for small stand-alone programs, while WTP is designed for multi-program applications. Both protocols provide a method to generate HTML pages in response to URL requests.

Our target application consists of many hundreds or thousands of programs, linked into large executable units called *application transaction processes* (ATPs), typically several megabytes large. This application will serve many users across a IP network, using the HTTP server as a connection point, and HTML as the screen presentation language.

Our fundamental requirements for implementing such a large-scale web-based application are:

- **Efficiency**: while CGI creates a new process for each URL request, we wish to reuse the same process for multiple requests, in series. This is significantly faster.

- **Construction**: a CGI program is essentially stand-alone; we wish to be able to build applications out of many programs, each handling one logical HTML page or 'screen'. This permits large, complex applications.

- **Session control**: the user establishes a logical connection when entering the main HTML page (typically a sign-on screen), and maintains this logical connection across a number of URL requests, until it is terminated or broken. This permits intelligent applications.

- **Context management**: an application program can maintain information about the user's work in progress. For instance, when the user is scrolling through a list of database records, context permits the application program to correctly handle an action like 'Next'. This permits rich applications.

- **Distribution**: a realistic application may become too large to handle as one executable unit; we wish to be able to break the application into multiple ATPs, without extra work

by the programmer. The WTP manager is responsible for locating a suitable ATP to handle a URL request.

- **Load balancing**: when particular application functions are heavily used, we want to be able to run more than one instance of the same ATP, either statically (by hand) or dynamically (following the flow and ebb of user activity).

- **Stability**: a realistic application has programs that crash, loop, or corrupt memory. Such programs may not compromise the stability of the overall application.

We can also note that we wanted a protocol that is easy to use, transparent, portable to any platform, any program language, and any HTTP server.

10.1.2 Why Invent A New Protocol?

We considered, and rejected the CGI, FastCGI, and xxAPI (ISAPI, NSAPI, ASAPI, WSX) server plug-in protocols.

- We have built CGI prototypes which include session control and context management. However, such prototypes are not efficient, and do not allow construction, distribution, or load balancing.

- The xxAPI plug-in protocols require highly-skilled developers, and do not allow distribution, load-balancing, or context management. We do not believe that a xxAPI can provide an efficient model for heavy data processing: when a database operation takes many seconds to complete, the entire web server is blocked during this period. An xxAPI application cannot be guaranteed to be either stable or portable.

- We looked at the FastCGI protocol from OpenMarket (www.openmarket.com); this tackles the issue of efficiency, but not session control, context management, or distribution: a FastCGI application is a single executable unit. We considered adding session control and context management to this protocol (as we did for CGI), but that still leaves the issue of distribution unresolved.

- We looked at various CGI-hybrids for building web applications; most of these are a mixture of CGI combined with a server process: each URL request is passed to a small CGI program that makes a connection to the application server, sends the request, waits for a response, then returns that to the HTTP server. We did not choose such a model for various reasons. Firstly, it still requires a new process for each URL request, which will always be inefficient at high loads. Secondly, it passes all requests to a single server, which must either be multithreaded (i.e. complex) or single-threaded (i.e. slow). Neither of these match our needs. However, it would be possible to implement WTP in this manner, if one did not want to write a HTTP server plug-in (see figure 4).

The design of WTP is, therefore, a combination of these existing web application protocols with a solid transaction processing system that is as powerful as existing mainframe transaction processors. We consider transaction processing to be an essential basis for any realistic large-scale application.

WTP is implemented by a WTP manager program. The WTP manager can be embedded into the HTTP server (for instance the iMatix Web Transaction Server supports WTP

directly); it can be built as an xxAPI plug-in; it can even be implemented as a FastCGI program, or as a CGI-hybrid program.

This figure shows WTP support built-in to the web server:

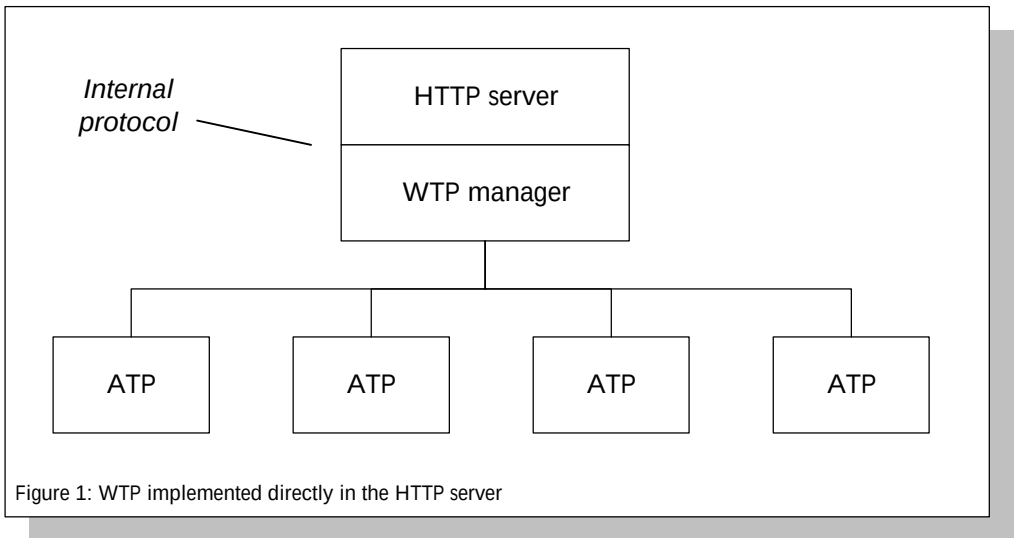


Figure 1: WTP implemented directly in the HTTP server

This figure shows WTP support built as a server plug-in:

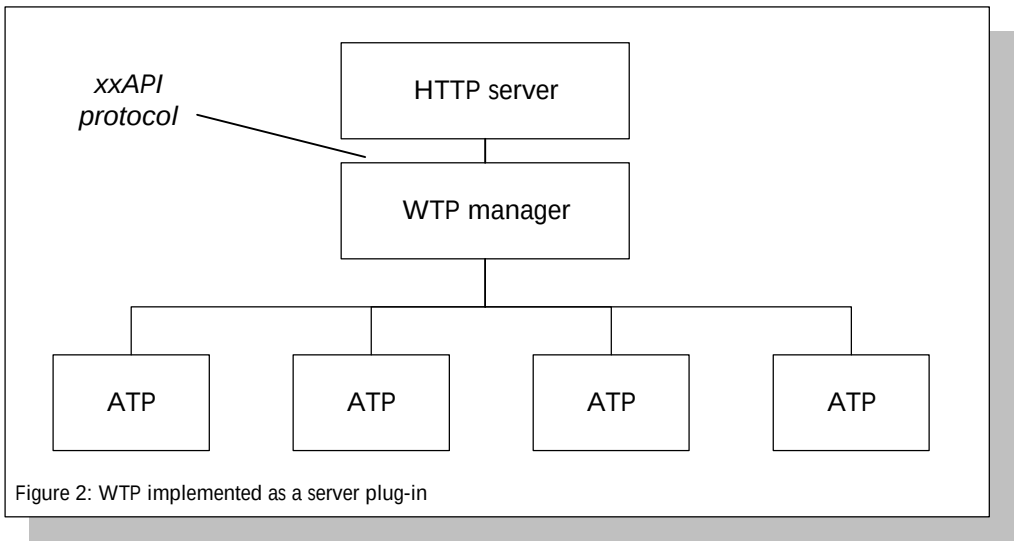


Figure 2: WTP implemented as a server plug-in

This figure shows WTP support built using FastCGI or a similar protocol:

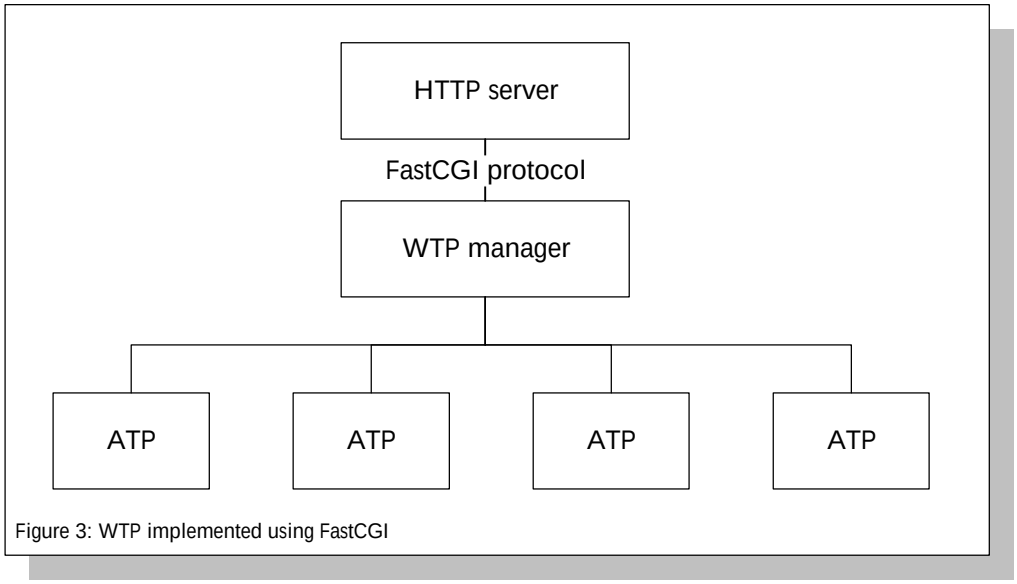


Figure 3: WTP implemented using FastCGI

This figure shows WTP support built as a CGI-hybrid program:

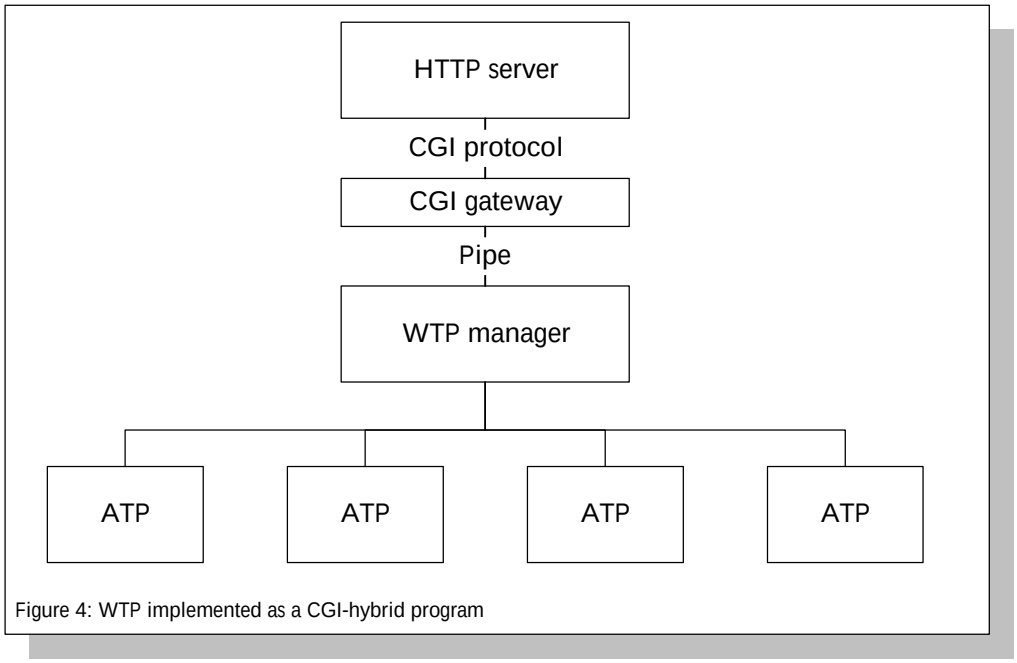


Figure 4: WTP implemented as a CGI-hybrid program

10.2 How WTP Works

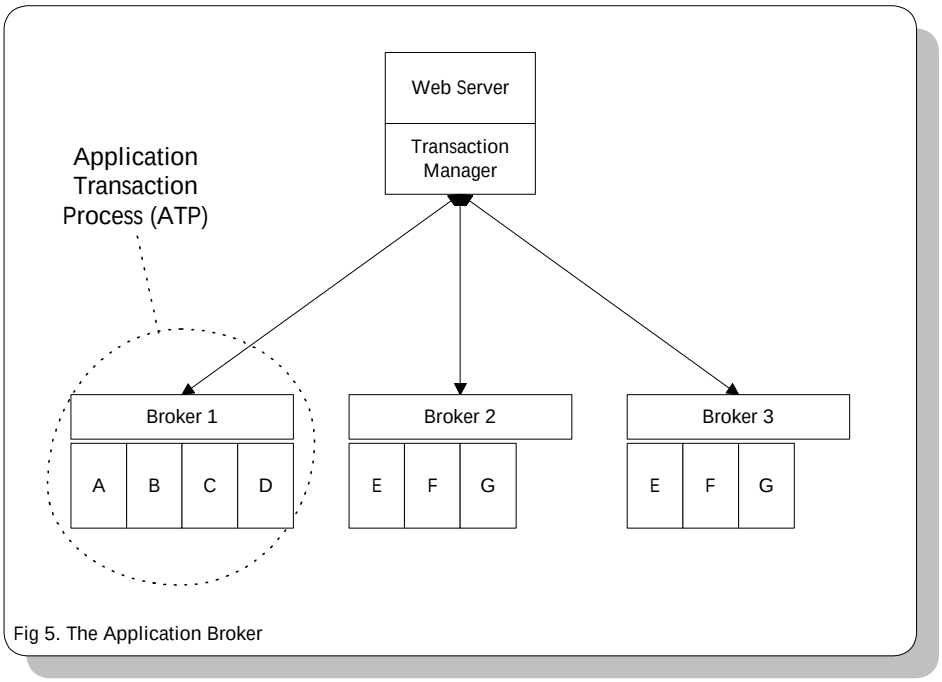


Fig 5. The Application Broker

The WTP manager is responsible for starting, monitoring, and halting ATP processes as required. One WTP manager (there may be several active on a particular host machine) is responsible for handling a set of WTP applications. For instance, one WTP manager could handle the development, test, and production versions of a client control application. Each of these applications can be stopped and started independently.

In practice we would use a separate server for production applications, to ensure the highest possible degree of stability and reliability.

Communications between the WTP manager and ATPs use a 'callback' mechanism as follows: the WTP manager creates a TCP or UPD port that accepts connection requests. When the WTP decides to start the application, it creates one process per ATP. The ATP starts-up, and connects to the WTP manager port. The ATP then 'registers' with the WTP manager, so that the WTP manager knows what work the ATP is able to do. The ATP then waits for requests from the WTP manager: when a request arrives, it handles it and responds with a reply. At any moment the WTP manager can choose to kill the ATP, or create further instances; equally the ATP can handle fatal errors by aborting if necessary.

10.2.1 ATP Initialisation

When the WTP manager starts an ATP, it passes a number of command-line arguments to the ATP main function:

- A VERSION string. This is "WTP/x.x" where 'x.x' is the WTP version number supported by the WTP manager. The ATP main function should check this string and take appropriate action. For example, if the ATP cannot handle the WTP version, it can write an error message to the stderr stream, and exit.

- A PROTOCOL specifier. This may be 'tcp', 'udp', or 'rdtp', and indicates which protocol the WTP manager can handle on its callback port. TCP is the best-known internet protocol; UDP is a simpler and faster protocol suitable to local connections; RDTP is an experimental protocol being developed by iMatix that combines the speed advantages of UDP with the reliability of TCP.

- A CALLBACK PORT number. This is specified as a string, e.g. "5500".

- A CALLBACK KEY. This is a string, e.g. "P83hXSb8AzyU", that the ATP must supply during connection. The purpose of the callback key is to ensure that only authorised ATPs try to connect to the callback port. The WTP manager generates a unique callback key for each ATP instance that it starts.

Using this information, the ATP connects to the WTP manager (by sending a WTP_CONNECT message), then registers a number of application programs by sending zero or more WTP_REGISTER messages.

Typically we build the callback and registration logic into the ATP main function. We call this the 'broker program'. Broker programs can be written by hand, or generated. The WTP toolkit includes tools to generate these programs, and function libraries to encapsulate much of the necessary work.

10.2.2 WTP Messages

WTP messages use a compact representation aimed at efficiency rather than readability. We did not choose the style of a HTTP message for two reasons. Firstly, HTTP messages are not explicitly sized, so cause difficulties for persistent connections. I.e. the original HTTP protocol assumed that the end of a message was equivalent to the end of a connection. Secondly, HTTP messages are quite verbose, an overhead that we wanted to avoid.

This is the format of a WTP message:

```
[message size]      4 bytes, in network order (hi to lo)
[message type]      1 byte, defining the message type
[message body]      zero or more fields
```

The message size specifies the size of the message excluding the two size bytes. The wtpdefn.h file defines a set of constants for C programs that use WTP. The message body consists of zero or more fields, implicitly defined by the type of message. Fields can be any of these types:

Field type:	Has this meaning:
byte	A 1-byte value
dbyte	A 2-byte value, in network byte order
qbyte	A 4-byte value, in network byte order
string	A null-terminated string
block	A block of data, specified as a four-byte size field plus a series of bytes. Not null terminated.

These are the WTP messages that an ATP can send to the WTP manager:

WTP CONNECT	Connects to the WTP manager
WTP REGISTER	Registers a program
WTP READY	Signal ready for work
WTP DISCONNECT	Disconnect from the WTP manager
WTP ERROR	Request failed
WTP DONESHOW	End program; show HTML screen
WTP DONECALL	End program; call new program
WTP DONERETURN	End program; return to calling program
WTP DONEEXIT	End program; exit the application
WTP DONEERROR	End program; there was a fatal error

These are the WTP messages that the WTP manager can send to an ATP:

WTP DO	Execute some program
WTP OK	Request suceeded
WTP ERROR	Request failed
WTP DISCONNECT	ATP should terminate

All messages are sent on the basis of 'question and response'. Invalid messages get a WTP_ERROR reply with the error code WTP_ERRORINVALID. The WTP manager may try to recover from an invalid message, or may break the connection.

10.2.2.1 The WTP_CONNECT Message

```
[message type]   byte    WTP_CONNECT
[callback key]   string  As supplied by the WTP manager
[signature]      qbyte   Version identification signature
```

This message must be the first message that an ATP sends to the WTP manager. It establishes the logical connection between the ATP and the WTP manager. The callback key is used by the WTP manager to ensure that only real and valid ATPs can connect. The signature string is a 32-bit value the ATP should generate from its executable file date and time. This is used to allow detection of incompatible or changed ATP executable versions.

The WTP manager responds to a WTP_CONNECT message with a WTP_OK or a WTP_ERROR message, with one of these error codes:

WTP ERRORUNAUTHORISED - An invalid callback key was supplied

WTP ERRORUNEXPECTED - Not allowed at this point

10.2.2.2 The WTP_REGISTER Message

```
[message type]   byte    WTP_REGISTER
[program name]   string  Name of program
[is root]        byte    1 if this is the root program, else 0.
```

This message tells the WTP manager which programs that the ATP is able to run. One program generally corresponds to a HTML screen. The ATP sends one WTP_REGISTER message for each program it contains. If no root program is specified, the WTP manager may reject any connection to the application with an appropriate error message. If several root programs are specified, the WTP manager may choose to use the first, the last, or use some other algorithm to decide which root program to launch. Typical applications will specific exactly one root program.

The WTP manager responds to a WTP_REGISTER message with a WTP_OK or a WTP_ERROR message, with one of these error codes:

WTP ERRORUNCONNECTED - WTP CONNECT was not sent, or failed

WTP ERRORUNEXPECTED - Not allowed at this point

The WTP_READY Message

```
[message type]   byte    WTP_READY
```

This message tells the WTP manager that the ATP is ready to accept application program requests. The WTP manager responds to a WTP_CONNECT message with a WTP_OK or a WTP_ERROR message, with one of these error codes:

WTP ERRORUNCONNECTED - WTP CONNECT was not sent, or failed

WTP ERRORUNEXPECTED - Not allowed at this point

The WTP_DISCONNECT Message

```
[message type]   byte    WTP_DISCONNECT
```

This message allows an ATP to terminate the connection to the WTP manager. This message is not strictly needed, since the WTP manager will detect that an ATP has terminated, and handle the disconnection automatically. The WTP manager does not respond to a WTP_DISCONNECT message.

10.2.2.3 The WTP_OK Message

```
[message type]   byte    WTP_OK
```

This message is sent as a positive response, and never receives a response.

10.2.2.4 The WTP_ERROR Message

```
[message type]   byte    WTP_ERROR
[error code]     dbyte   Cause of the error, as a numeric code
[error reason]   string  Cause of the error, as a string
```

This message is sent as a negative response, and never receives a response.

10.2.2.5 The WTP_DO Message

```
[message type]   byte    WTP_DO
[signature]      qbyte   Version identification signature
[program name]   string  Program to execute
[entry code]     byte    Program entry code
```

```
[HTTP URI]        string  URI for use in HTML hyperlinks
[HTTP data]       string  Encoded HTTP query data, if any
[arguments]       block   Program call arguments, if any
[call result]     byte    Call result indicator
[environment]     block   HTTP environment block
[global context] block   Global context block
[local context]  block   Local context block
```

This message asks the ATP to execute a specific program. The entry code can be one of:

WTP DOINIT Initial entry into the program

WTP DOGET Program has to process HTML form data

WTP DOCONTINUE A called program finished its work

The use of the entry code is explained in the section "The WTP Program Model".

The signature is that supplied by the ATP at connection time. The ATP should recalculate the signature, and if it fails to match, return a WTP_ERRORSIGNATURE code.

The HTTP URI must be used by the application programs when they create HTML links in their HTML screens. The URI is encoded to contain a 'session key', i.e. information that the WTP manager needs to identify the session when the user uses an action on the HTML form. The HTTP URI is explained in the section "WTP Session Control".

When the program execution state is WTP_DOGET, the HTTP data string holds the encoded HTTP form or query data. Otherwise this string is empty (a single null byte). The format of this data is explained in the section "HTTP Form Data Encoding". The call arguments block is empty.

When the program state is WTP_DOINIT, the call arguments block holds the arguments supplied by the calling program. If the program being executed is the application root program (i.e. it has no calling program), then the call arguments may be empty, or may contain any 'command line' arguments specified by the user in the URL which invokved the WTP application.

When the program state is WTP_DOCONTINUE, the call arguments block holds the return arguments from the called program, and the call result indicator is set to one of the values listed below.

These are the possible values for the call result indicator:

WTP NOERROR Call succeeded

WTP ERRORNOTFOUND Requested program is not known

WTP ERRORWOULDLOOP Requested program is already active

WTP ERROROVERFLOW Maximum number of active programs reached

The HTTP environment block contains the HTTP header fields and standard CGI variables (like REMOTE_HOST). This block is only supplied to the application root program when it starts, since it is essentially identical for all WTP_DO messages for a session. At other times this block is empty.

The ATP responds to a WTP_DO message with WTP_DONExxxx if there were no problems, or WTP_ERROR if there was a problem, with one of these error codes:

WTP ERRORNOTFOUND The program is not known

WTP ERRORUNAVAILABLE The program is no longer available

WTP ERRORSIGNATURE The ATP signature has changedW

TP ERRORUNEXPECTED Not allowed at this point

10.2.2.6 The WTP_DONESHOW Message

```
[message type]   byte    WTP_DONESHOW
[HTML data]      string  HTML screen data
[global context] block   Global context block
[local context]  block   Local context block
```

An application program has finished a logical unit of work when it (a) is ready to display a form, (b) wants to calls another application program, or (c) has terminated, either normally, or following some error. In the first of these cases, it returns a WTP_DONESHOW message.

10.2.2.7 The WTP_DONECALL Message

```
[message type]   byte    WTP_DONECALL
[program name]   string  Program to call
[arguments]      block   Arguments for called program
[global context] block   Global context block
[local context]  block   Local context block
```

This message tells the WTP manager to call a new program. The current program is suspended, and will resume only when the called program sends a WTP_DONERETURN message. No HTML is sent to the user at this point; the WTP manager must locate and start the requested program.

10.2.2.8 The WTP_DONERETURN Message

```
[message type]   byte    WTP_DONERETURN
[arguments]      block   Arguments back to parent program
[global context] block   Global context block
```

This message tells the WTP manager to return to the previous parent program. If the current program was the root program, this message is treated as a WTP_DONEEXIT message.

10.2.2.9 The WTP_DONEEXIT Message

```
[message type]   byte    WTP_DONEEXIT
```

This message tells the WTP manager to end the application session.

10.2.2.10 The WTP_DONEERROR Message

```
[message type]   byte    WTP_DONEERROR
[error reason]   string  Cause of the error, as a string
```

This message tells the WTP manager to end the application session, and show an error message to the user.

10.2.3 The WTP Program Model

The WTP program model enforces a transaction-based model. This was a deliberate design decision: our long experience in building successful large-scale business applications has taught us that this is a good way to build efficient, cheap, and robust applications.

These are the main differences between a 'normal' program and a WTP program:

1. The WTP program must send all its data to the client screen in one operation. Furthermore, this action is fused to the end of the transaction. There is no way for the program to display some data, wait for some input, and so on. This is a model that is well-known to CGI programmers, but less evident to Windows and UNIX programmers. In short, WTP uses the standard HTTP 'thin client' model.

2. WTP transaction ends when the program decides to display its HTML page. At this time, the database transaction (if any) is closed; all outstanding database requests are either committed or rolled-back; any open files are closed, and any temporary memory is released. A WTP transaction cannot remain 'open' while the user inputs data, for several reasons. Firstly, database resources may never be locked for more than a few seconds at most, to avoid deadlocks. Secondly, since WTP permits multiple instances of an ATP for load balancing, any process-specific resources (dynamically-allocated memory, open files,...) cannot be guarantied to be available when the program continues processing after receiving the form.

3. The actions of showing the HTML page, calling another program, or returning to the caller program are formalised and handled by the WTP manager, not the program. Again, this is necessary given the WTP distribution and load-balancing functions.

4. WTP program is invoked in different ways depending on the situation. The WTP_DO message uses WTP_DOINIT when the program is newly activated. It uses WTP_DOGET when the program is re-activated to handle HTTP form data. It uses WTP_DOCONTINUE when the program is re-activated after a called program ended.

5. Similarly, a WTP program must signal its intentions to the WTP manager. It does this by using different messages. WTP_DONESHOW means it wants to display an HTML page. When the user uses some action on the HTML page, the same program is re-activated with a WTP_DOGET entry code. WTP_DONECALL means it wants to call another program. WTP_DONERETURN means it has finished. WTP_DONEEXIT means it has decided to end the user session.

10.2.4 Walkthrough Of A WTP Transaction

Here we show the transactions involved in a typical operation, user sign-on. We show the principal sign-on screen, accept a user sign-on, and show a top-level menu screen. Finally we return to the sign-on screen:

- The WTP manager receives a user URL request for the application. It determines the main program, and sends a WTP_DO + WTP_DOINIT message to the appropriate ATP.

- The main program prepares the sign-on form, clears the user-name and password fields, and returns WTP_DONESHOW.

- The WTP manager - via the web server - displays the HTML page and waits for user input.

- The user enters data into the user-name and password fields, then clicks on the 'Sign-on' button. The web browser now sends the form data back to the web server, which passes it to the WTP manager.

- The WTP manager decodes the form data to extract a session key. Armed with this, it calls the main program once again with WTP_DO + WTP_DOGET.

- The main program decodes the HTTP form data, verifies the user name and password, and if it accepts them, decides to call the top-level menu program. It returns WTP_DONECALL.

- The WTP manager locates the ATP for the top-level menu program, then sends WTP_DO + WTP_DOINIT to the ATP.

- The menu program prepares its screen and returns WTP_DONESHOW.

- The user clicks on the 'Exit' button. The menu program receives the WTP_DO + WTP_DOGET, and returns WTP_DONERETURN.

- The WTP manager now sends a WTP_DO + WTP_DOCONTINUE back to the main program, which eventually replies with a WTP_DONESHOW.

10.2.5 WTP Session Control

The WTK manager is responsible for creating and managing the WTP session. There are many possible ways to do this; the choice of design is transparent for WTP applications; we describe one possible implementation, and our reasons for choosing it.

HTTP is a stateless protocol, but there are a number of ways to add state to a HTTP conversation. One common technique is 'cookies'. These are small strings of data that the server returns with a page. The browser will include these back in any later response. Unfortunately, cookies are often (mis)used as a technique to track user's access to a particular site; as a result many people disable their browsers' cookie functions. Another technique is to use hidden form fields. These fields are returned with the form when the user clicks on an action. Hidden form fields work well when all actions on a HTML page are implemented as submit buttons. There are cases, however, where this is cosmetically unacceptable. One example is where the user can make a selection from a list of client records. Such a list looks and works much better using hyperlinks. However, browsers do not interpret hyperlinks as form submission actions. (This can be programmed in JavaScript, but painfully, and -- to our knowledge -- only on one version of one browser, and that thanks to a bug.) The last candidate technique is to encode the session information in the URI used in hyperlinks. This requires that at the moment the HTML page is generated, the encoded URI be inserted into hyperlinks, along with other data sufficient to allow the application to use the resulting 'click'. An encoded URI could looks like this: "/wtp/application/?session=XYZ123". However, the WTP manager can choose any suitable encoding it likes, since it is solely responsible for decoding the URI.

The WTP manager supplies a suitable URI each time it sends a WTP_DO message to an ATP. This URI must at least specify the WTP application so that a hyperlink returns correctly to the WTP manager. If the WTP manager implements state using cookies, for instance, it must still supply a valid URI to the ATP.

10.2.6 The WTP URL Format

The format of a WTP application URL is:

```
http://hostname[:port]/wtp/application[?arguments]
```

The application can be specified as one or more levels, e.g.:

```
http://www.imatix.com/wtp/clients/dev/
```

10.2.7 Context Management

The WTP_DO and WTP_DONExxxx messages include two blocks called the 'global context' and 'local context'. The global context block is an area of memory that is shared between all programs in a session. This can be used to store information that is pertinent to the whole session, for instance information about the user. The global context block is initialised as an empty block (size zero) when the session is created. All WTP_DONExxxx messages update the global context block.

The local context block holds information for the current program only. The WTP manager initialises this block when starting a new program (either the root program or following a DONE_CALL). It deletes the block when the program terminates (DONE_RETURN).

10.2.8 HTTP Form Data Encoding

The HTTP form data encoding format (sometimes called 'MIME' encoding) is identical to that provided to CGI programs on their stdin stream or on their command line. The HTTP data consists of a series of encoded 'name=value' pairs, separated by & or ; characters. Each 'name=value' pair is encoded using the following escape mechanism: all characters except alphanumerics and spaces are converted into the 3-byte sequence "%xx" where xx is the character's hexadecimal value; spaces are replaced by '+'. Line breaks are stored as "%0D%0A", where a 'line break' is any one of: "\n", "\r", "\n\r", or "\r\n". The WTP support libraries provide functions to decode and access such data strings.

A WTP application will typically be driven by HTTP POST operations (in which data from a form is posted) and by HTTP GET operations (typically the result of hyperlinks or direct requests to a page). In general, a POST can only be done through a push-button or image; a GET can be done through a hyperlinked text or image.

With suitable encoding, a GET operation will return data that can be used much as POSTed data. To allow the WTP application to detect that data was provided by GET arguments rather than through a POST, we use the convention that GET argument data starts with '&'. This extra character can be skipped by the HTTP decoding routines.

10.2.9 Support for National Character Sets

The HTTP form data can be encoded using the SGML meta-characters for non-portable national characters. However, the WTP manager will do a reasonable attempt to translate characters where it can. It will do this on output only.

11. Abbreviations and Terminology

Application: A large piece of software that provides a set of functions, typically built around a business database. A small applications might work with a few dozen database tables; a large application with several thousand. An application consists of a number of application programs. WTP is concerned primarily with 'business applications', also called 'commercial data processing'.

ASAPI: The protocol provided by Apache web servers for internal add-ons.

ATP: Application transaction process; this is an executable unit consisting of one or more application programs managed by a broker. A WTP application is organised into one or more ATPs, for reasons of convenience and tuning.

Broker: A program responsible for handling the WTP protocol for an ATP. Broker programs can be written by hand, or generated from templates.

Client: In general, the party which sends requests to a server and waits for answers. We tend to describe distributed systems in terms of clients and servers, since this model is generally better understood than a system where any party can be either a client or a server.

Context: Information that programs need to save and restore in order to be able to continue working on behalf of a specific user session. Since the same program can be used serially by many sessions, it cannot rely on its own memory to hold session-dependent data.

Intranet: A local-area network based on TCP/IP.

ISAM: Indexed-sequential access method; also called VSAM on IBM mainframes.

ISAPI: The protocol provided by Microsoft web servers for internal add-ons.

NSAPI: The protocol provided by Netscape web servers for internal add-ons.

Program: A component of a software application. There will be many types of program, but the most important for the application developers are programs that handle specific screens. We tend to associate one screen with one program, for simplicity. One consequence of using WTP is that the breakdown of an application into 'programs' becomes formalised. This is deliberate.

Screen: One HTML page, representing a specific application function. For example, a 'Client Search' screen might provide functions to search through a list of clients.

Server: A program that handles requests from a set of clients. We speak of a web server, which handles HTTP ('web') requests, and an application server that handles requests for specific application programs.

Session: One user's logical connection to the application. In a typical session, a user will sign-on, do some work, then sign-out. A session may last for hours. A user can open many

sessions at once. The WTP protocol is responsible for keeping these sessions separate from each other.

Transaction: A logical unit of work, which usually culminates in the display of some data as an HTML page.

Transaction: Used to mean many things, but generally means the work involved in handling a client request. For instance, when a users modifies a database value, the transaction usually involves locking the record, making the modification, and committing the changes. From the user's perspective, a transaction starts when they request something from the server, and ends when the reply comes back and is shown on the screen.

URI: Uniform resource indicator; a URL without a host specified. E.g. /wtp/demo.

URL: Uniform resource locator, e.g. http://www.imatix.com/wtp/demo

WSX: Web server extensions; the protocol provided by the *Xitami* web server for internal add-ons.

xxAPI*:* A generic term for any of ASAPI, NSAPI, or ISAPI.