Ostia Portus

Concepts

Version 2012-12-17

December 2012

Portus

This document applies to Ostia Portus 2012-12-17 15:50:18 (MET) and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

© Copyright Ostia 2012.
All rights reserved.

The name Ostia Software Solutions and/or all Ostia Software Solutions product names are either trademarks or registered trademarks of Ostia Software Solutions. Other company and product names mentioned herein may be trademarks of their respective owners.

Table of Contents

1 Introducing SMARTS 1
The SMARTS Concept 2
Who uses SMARTS? 3
Why use SMARTS? 3
The SMARTS Server Environment 5
Architecture 6
Supported Environments 7
How does SMARTS work? 10
SMARTS Control Blocks 15
Implementing a SMARTS Application 16

1 Introducing SMARTS

- The SMARTS Concept 2
- Who uses SMARTS? 3
- Why use SMARTS? 3
- The SMARTS Server Environment 5
- Architecture 6
- Supported Environments 7
- How does SMARTS work? 10
- SMARTS Control Blocks 15
- Implementing a SMARTS Application 16

The same Software AG product may run in a number of environments, each based on a different architecture. When a product developed for one environment needs to run in another environment, the product must be "ported" to the new environment -- a costly and time-consuming process that must be repeated for every new level of the product.

The Software AG Multiple Architecture Runtime System (SMARTS) provides an application programming interface (API) for porting that is independent of any particular environment. Individual environments are mapped to this API only. Software AG products that interface with the API are thus automatically ported to all supported S/390 mainframe environments, saving the time and costs previously required for porting.

SMARTS is the runtime layer for a number of environments available on IBM, Fujitsu Siemens Computers, and Fujitsu S/390 mainframe operating systems. In each environment, SMARTS implements a POSIX-like layer for running POSIX-like programs.

This chapter covers the following topics:

- The SMARTS Concept
- Who uses SMARTS?
- Why use SMARTS?
- The SMARTS Server Environment
- Architecture
- Supported Environments
- How does SMARTS work?
- SMARTS Control Blocks
- Implementing a SMARTS Application

The SMARTS Concept

Traditionally in business environments, software products were developed on S/390 mainframe computers. The code in early programming languages was difficult to maintain and the products often required the user to have a significant level of training. Thus the cost of using and maintaining the products was high.

The advent of personal computing has brought higher level programming languages that are easier to write and maintain, and graphical user interfaces that make the products easier to use. However, PCs will not replace the S/390 mainframe environment:

- PCs are not as reliable as S/390 mainframe systems, which can run 24 hours a day; 7 days a week; 52 weeks a year.
- PCs cannot compete with the scale and performance of S/390 mainframe runtime systems.

For most large enterprises, the most cost-effective solution is server-centered with an S/390 mainframe providing the services.

Because these issues are so important to large businesses, it is clear that S/390 mainframe computers are still needed. But porting PC products onto S/390 mainframe systems requires a costly duplication of effort and delay in implementation.

When the SMARTS API is used, products no longer need to be individually ported to multiple architectures. If an application runs on SMARTS, it automatically runs in all supported S/390 mainframe environments.

Who uses SMARTS?

Application or product developers who wish to target the operating systems and environments supported by SMARTS need to program to the POSIX-like API.

As SMARTS can support C, Natural, Assembler, COBOL, and similar high level languages, it can be used by both

- product development groups using these programming languages; and
- user installations developing applications in the supported programming languages.

The following figure illustrates SMARTS as the Software AG Application Server:

Construct	XML-based application	Existing code
Natural support	Tamino	HLLI support
Transaction, System, and Security Management, Messaging; HTTP Connectivity, Naming and Transport Services		
SMARTS		

Why use SMARTS?

As a host environment, SMARTS provides the benefits outlined in the following sections.

- Portability of Code
- Speed to Market
- Layered Design
- Common Device Interface
- Focused Development Effort

- Focused Support Effort

Portability of Code

Once C programs are ported to the SMARTS API, they can be run in any other appropriate SMARTS C run-time environment by simply relinking the C code. For applications developed in programming languages other than C, SMARTS provides high-level language extensions that enable you to call the POSIX-like API from Natural, Assembler, COBOL, and similar high level languages.

By removing the cost of porting to particular environments, SMARTS makes it possible to run an application in lesser-used environments where otherwise it would not be cost-effective to do so.

Speed to Market

Software developed on Open Systems or Windows NT is easily run on the SMARTS platform due to the implementation of a POSIX-like interface. Once ported to the SMARTS platform, object code can simply be relinked to run on any supported SMARTS platform.

Ports to the S/390 mainframe are quicker and since little additional porting work is required, rollout of the product on less-strategic S/390 mainframe platforms follows immediately. Products can appear on SMARTS-supported S/390 mainframe platforms much more quickly with SMARTS than without.

Layered Design

The layered structure of SMARTS means that code is split into

- independent code, which forms the majority of the SMARTS nucleus; and
- dependent code paths that are driven (or drive the nucleus) using well-defined interfaces.

Platform experts can make the best use of the underlying operating system technology while applications running on SMARTS see exactly the same interface layer on all platforms. This leads to consistent code paths and robust code that can be reused by any products that run on SMARTS.

Common Device Interface

The common device interface (CDI) provides a "plug and play" technology that enables the SMARTS independent nucleus to communicate with any subsystem available on a given operating system without modifying the application. CDI also allows the creation of logical subsystems for supporting in-store files, pipes, and so on, using the most appropriate operating system services available.

Focused Development Effort

Development resources to support S/390 mainframe operating systems are characteristically limited and therefore must be used efficiently. Well-defined interfaces make it possible for an expert on any given operating system to implement code and communicate with any subsystems, no matter what group they belong to within the company. By integrating this code into SMARTS, the development efforts of one company group become available to all groups. Focusing development resources in this way means cleaner code that can be supported and improved as time goes on.

Focused Support Effort

In the same way as development, support benefits when more than one product is using the underlying code base. Fixes created for one product benefit all other products using the same underlying SMARTS technology, which leads to a more robust run-time environment for all products.

The SMARTS Server Environment

SMARTS is a runtime system that provides:

1. a server environment on all platforms
2. one or more client environments for each supported platform.

The SMARTS server environment provides a multitasking architecture that uses blocks of storage called "threads" in which SMARTS application programs can run. By ensuring that all application storage is in contiguous blocks of the address space, the SMARTS server environment is in a position to move dormant applications out of the address space to make room for other more active users. If or when such a dormant application becomes active again, it is simply moved back into the address space and dispatched again.

The SMARTS server environment

- uses its multitasking structure to make full use of operating system subtasks to drive the system CPUs.
- uses contiguous storage threads to give it more control over the applications' storage areas, thus enabling it to handle more applications running in the one address space.
- shares the underlying operating system subtasks between users and is thus not subject to restrictions on the number of processes that can be concurrently supported.
- uses a state-of-the-art buffer pool management technique that ensures consistent path lengths, no fragmentation of storage areas, and expansion and contraction of the storage areas within the address space.

- uses a state-of-the-art resource manager that ensures the shortest path length possible for the serialization of resources. The technique uses only machine instructions unless it is necessary to wait for a resource.
- is ready now for 24-by-7 operation as the nucleus does not need to be cycled for any internal reasons and the buffer and resource managers can handle higher than expected loads if the resources are available in the address space.

Architecture

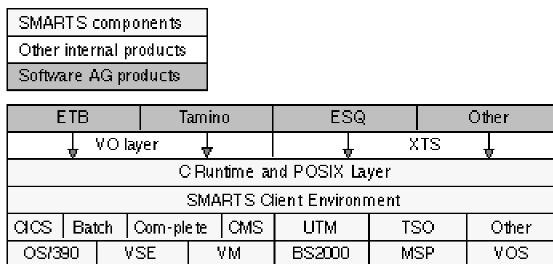
The SMARTS architecture has two distinctly different views: client and server.

Client

SMARTS supports client programs and interfaces in the various client environments such as Complete, CICS, and batch.

In many cases, the client environment is required to support an interface provided by a Software AG product. Using a server product as an example: the primary code runs in the SMARTS server environment but access to the services is normally provided from all appropriate client environments.

An API is generally provided in the various client environments. Such APIs are supported by the SMARTS client environment. The various client environments are detailed later in this document.



SMARTS Client Structure

Server

The SMARTS server environment

- runs high availability, high throughput server applications.
- includes built-in support for Software AG's System Management Server to uniformly manage and control the services running within the environment.
- supports all POSIX interfaces supported by the SMARTS POSIX layer. The client environment, on the other hand, may not support all functionality or support it less efficiently than the server environment.

SMARTS components
Other internal products
Software AG products

ETB		Tamino		ESQ	Other
VO layer		SMP API		XTS	HTTP
C Runtime and POSIX Layer					
SMARTS Server Environment					
SMARTS Context Manager					
OS/390	VSE	VM	BS2000	MSP	VOS

SMARTS Server Structure

Supported Environments

The strength of SMARTS lies in the number of operating systems and environments it can support. SMARTS provides the server environment on every supported operating system while supporting client environments on those operating systems according to the requirements of the SMARTS-based applications.

Where an application is fully C-based, it is possible to compile the C code once to provide a SMARTS C system 390 object code base that is then simply relinked to run in any of the environments listed below.

Note: The following list describes what SMARTS may support and does not commit Software AG to supporting those environments. Consult your local Software AG representative, or the Software AG web site for the platform and environment support currently available.

- Operating Systems

- Batch and TP Monitor Support across Platforms

Operating Systems

It is intended that SMARTS will support the following operating system environments on all version levels supported by their respective vendors:

- OS/390 (IBM)
- VSE/ESA (IBM)
- VM/ESA (IBM)
- BS2000/OSD (FSC)
- OS IV/F4 MSP (Fujitsu)

Contact Software AG for a list of the version levels currently supported.

Batch and TP Monitor Support across Platforms

SMARTS currently runs in batch or under the Software AG S/390 mainframe TP monitor environment Com-plete. Com-plete version 6.1 is based on SMARTS and therefore fully supports SMARTS-based clients.

OS/390 Operating System

OS/390 is the most strategic of all S/390 mainframe operating systems. Under normal circumstances, a SMARTS-based application is ported to this operating system first and then relinked to run on the other supported operating systems. This makes use of the many porting facilities available on OS/390 and within the Open Edition environment.

The following SMARTS environments are supported on OS/390:

- The SMARTS server environment
- Batch client
- Com-plete client
- CICS client
- IMS client

VSE/ESA Operating System

VSE/ESA has a strong base of customers who want to continue using it. Using SMARTS, Software AG can easily provide customers on this platform with the latest technologies.

The following SMARTS environments are supported on VSE/ESA:

- The SMARTS server environment
- Batch client
- Com-plete client
- CICS client

VM/GCS Operating System

The SMARTS server environment is supported on VM/GCS.

BS2000/OSD Operating System

Like VSE/ESA, BS2000/OSD has a strong base of customers who want to continue using it. Using SMARTS, Software AG can easily provide customers on this platform with the latest technologies.

The following SMARTS environments are supported on BS2000/OSD:

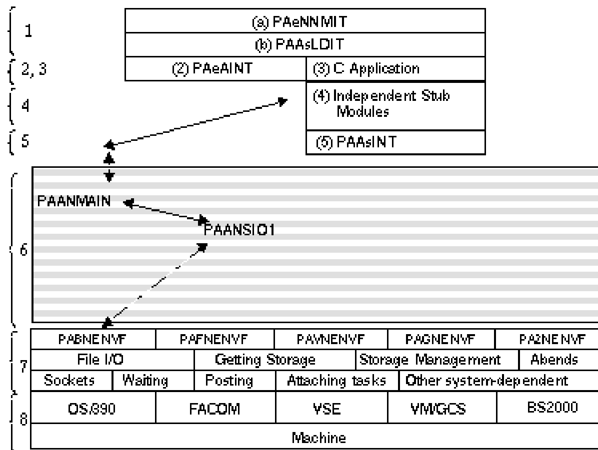
- The SMARTS server environment
- Batch client
- Dialogue client

OS IV/F4 MSP (FACOM) Operating System

The following SMARTS environments are supported on Fujitsu's MSP operating system:

- The SMARTS server environment
- Batch client
- Com-plete client

How does SMARTS work?



Naming Conventions

The module names used in the **diagram** have the following format:

- The first character of all example modules refers to the particular server component:

P	POSIX API component
R	anything else that relates to the SMARTS server component

- The second character identifies the program type:

A	Assembler programs
J	Job control programs
M	Assembler macro programs

- The third character indicates the environment used:

A	all environments
B	IBM's OS/390 or MVS/ESA batch or generic member
C	IBM's OS/390 or MVS/ESA CICS specific member
F	Fujitsu's FACOM OS IV/F4 MSP batch or generic member
G	IBM's VM/GCS batch or generic member
I	IBM's OS/390 or MVS/ESA IMS specific member
V	IBM's VSE/ESA batch or generic member

2	Fujitsu Siemens Computers' BS2000/OSD batch or generic member

All TP monitor options are included for each operating system. For example, "B" (OS/390 or MVS/ESA) includes batch, Com-plete, CICS, and OS/TS; "V" (VSE/ESA) includes batch, Com-plete, and CICS and OS/TS.

- The fourth letter indicates the subsystem to which the member is related. For example:

S	indicates the SAS/C compiler environment
N	indicates the server nucleus
Y	indicates the System C compiler environment

The remaining letters are unique and should indicate the actual function of the module:

INT	interface module
ENVF	module containing environment-specific functions
DSM	data space management functions
UXIT	user exit

Overview Diagram

The previous overview **diagram** presents SMARTS operation in terms of layers:

1	The SMARTS Wrapper	
a	Environment-specific initialization / termination logic	
b	Language- / runtime-specific initialization / termination logic	
2	Environment-specific low-level interface module	
3	Application written in the C language	
4	Environment-independent function stubs	
5	Language- / runtime-specific interface module	
6	Nucleus (independent)	
7	System management (environment-dependent)	
8	Machine (environment-dependent)	

The diagram shows an example flow through the system. The full lines show mandatory flows, while the broken lines show optional flows.

Each environment has a number of modules that are only used in that particular environment. In the example, these modules are

- PAeNMNIT, which does the environment-specific initialization and termination processing for a program; and

- PAeAINT, which is the low-level interface module for the environment used to access the SMARTS nucleus to initialize and terminate the environment.

Examples of these modules are PABNMNIT and PABAINT, which are used for the OS/390 batch environment.

To start, a C application calls the `printf()` function, for example.

The C header files, through the stubs, provide the entry to the environment-dependent initialization layer at PAAsINT. The stub module refers only to the initialization module specific to the environment it is running on.

The call then enters the independent layer through PAANMAIN, which directs it to the relevant function slot PAANSIO1 in the independent nucleus.

Up to this point the flow has been mandatory.

Depending on the tasks required for the application to work, the environment-dependent system management layer may need to be used. This is reflected with an optional flow from PAANSIO1 to PAeNENVF in the diagram.

The SMARTS Wrapper

The C application is actually wrapped by two SMARTS modules:

- PAeNMNIT is an environment-dependent module that takes control from the environment and builds a standard structure for the following modules to use. It then passes control to PAAsLDIT.
- PAAsLDIT is a compiler- or language-dependent module that completes the initialization of the environment and passes control to the actual C code.

The SMARTS Environment-Dependent Interface Module

PAeAINT, the SMARTS environment-dependent interface module, is normally called only for environment initialization and termination when the language-dependent interface module cannot be used.

C Application

Code for an application in C must be written to the C POSIX standard, a widely accepted UNIX standard for C.

When a C program is compiled, it will include standard headers. The C header files include prototypes for all the functions defined to SMARTS that a program may use. Using the #pragma preprocessor directive, the header files map each function name to a name internal to SMARTS that begins with "PS" and is followed by six character identifying the function uniquely.

For example, the `printf()` function is mapped to the SMARTS internal name PSPRINTF as follows:

```
#pragma map (printf, "PSPRINTF")
```

When compiled and linked, the C program has a number of external references to these PSxxxxxx objects. SMARTS provides these objects which are in fact simple Assembler stub modules that provide an interface to the SMARTS POSIX nucleus. The stubs are designed to enter the SMARTS nucleus with an indication of the function the user program has requested.

The stub contains a code that is a reference point to its particular slot within the independent nucleus. The stub also allows the application to enter SMARTS using a branch to a V constant (PAANINT), which is an external reference resolved at runtime. PAANINT is an entry point in each language- / runtime-specific interface.

The Independent Stub Modules

Each function used by the application is represented by a small environment-independent stub module.

The Compiler- or Language-Dependent Interface

PAAsINT, the compiler- or language-dependent interface module, provides an interface based on language- or compiler-dependent constructs to the SMARTS nucleus to quickly issue function calls from the application.

Environment-Independent Nucleus

The environment-independent layer contains pure Assembler code that can run on any S/390 mainframe. PAANMAIN is entered with a function code passed from the stub. Once the function code is validated, it provides the offset into the independent nucleus for the function to be serviced in SMARTS.

Environment-Dependent System Management

Environment-dependent modules are called to carry out tasks that are specific to the environment that the application is running on.

These functions include: file I/O processing, sockets processing, storage management, waiting, posting, ABENDs, attaching tasks and any other functions that are environment-dependent.

Example

The **diagram** above can be used to illustrate the life of the `printf` function in SMARTS:

1. The SMARTS environment-dependent initialization module PAeNMNIT receives control and builds the standard SMARTS environment.
2. Control is passed to the compiler- or language-dependent layer PAAsLDIT, which issues an ENVINIT call to initialize the environment.
3. As the runtime has not yet been built, control is then passed to the environment-dependent interface module PAeAINT, which in turn passes control to the PAANMAIN function in the independent nucleus to satisfy the request.

   If it is the first time through for the function, the environment is initialized:

   - all the environment-dependent information is loaded;
   - the POSIX program function table PGMT is read; and
   - all the addresses are put into the POSIX main control block PMCB.

     The PMBNINTP, PMBNGANC, PABNKERN, and PAANSERV modules are involved.

   If the function has been through before, the environment is already initialized and the function passes on to the environment-independent layer.
4. When the environment has been successfully built, the C program receives control.
5. A C application invokes the `printf` function.

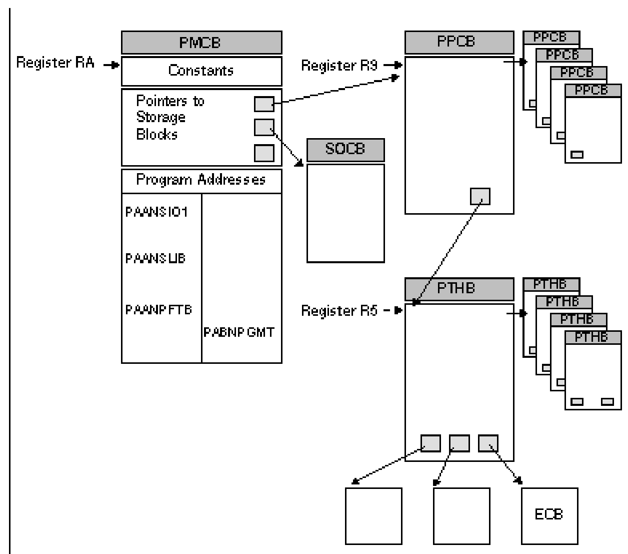
   The function is mapped to the PSPRINTF stub, which branches to the V constant PAANINT, the entry points to the SMARTS nucleus.
6. **Environment-dependent layer:**
   Assuming a System/C compiler environment, the branch is to PAAYINT.
7. **Environment-independent layer:**
   The function passes to the PAANMAIN module, which checks the POSIX function table (PFTB) to match the function code and obtains the offset into the POSIX main control block (PMCB) for the function. The function is passed to the PAANSIO1 module, which determines whether any environment-dependent system management tasks are required.
8. **Environment-dependent layer:**
   The environment-dependent module for an MVS environment PABNENVF is called to perform any required system management tasks.
9. **Module termination:**
   When the C main program returns to the language- or compiler-dependent module PAAsLDIT, the environment is terminated using the PAeAINT interface and control is returned to the operating system through the PAeNMNIT module.

SMARTS Control Blocks

Controlling the Process of Work

The main control block of the SMARTS environment is PMCB. Its process control block is PPCB and its thread or 'pthread' control block is PTHB.

The PMCB, PPCB, and PTHB control blocks together control the process of work.



SMARTS Environment Control Blocks

PMCB - POSIX Main Control Block

PMCB is the primary control block in the SMARTS system and represents the base control block for the SMARTS kernel. It is the first control block to be allocated by the environment-dependent kernel interface during kernel initialization and the last control block to be freed during kernel termination.

The control block is generated using the PMANPMCB macro.

PPCB - POSIX Process Control Block

The POSIX process control block (PPCB) represents the logical UNIX-like process in SMARTS and is the base control block related to a given SMARTS context. All context-related data is allocated in or chained off this area. The PPCB is allocated in global storage the first time a SMARTS context issues a SMARTS call. It is allocated in global storage as by default it will be chained from the PMCB.

The PPCB is generated using the PMANPPCB macro.

PTHB - pthread (POSIX Thread)

The pthread represents the fundamental unit of work within SMARTS. The POSIX thread control block (PTHB) contains pointers to any storage that is relative to the piece of work, which includes pointers to various event control blocks (ECBs).

The PTHB is generated using the PMANPTHB macro.

Information Control Blocks

PAANPFTB - POSIX Function Table

The POSIX function table control block (PAANPFTB) contains information about each function:

- name of the function; for example, `printf()`
- offset into the PMCB of the address of the program that implements the requested function
- input parameters
- return values

PABNPGMT - Program Table

The POSIX program table control block (PABNPFTB) contains a list of modules to be loaded; for example, PAANSIO1.

Implementing a SMARTS Application

The effort required to implement an application on SMARTS depends on the programming language used. The following sections describe the restrictions and efforts required in each case.

- Programming Language Used
- C-based Programs

- Installation and Maintenance

Programming Language Used

SMARTS provides the best possible support for any chosen programming language.

Given a programming language, the effort required from SMARTS to run the application or product on other SMARTS environments is discussed in the following paragraphs.

C-based Applications

Because SMARTS essentially replaces the run-time environment for a C-based application, one of the supported C compilers must be used. Currently, support is provided for the System/C compiler, which is the strategic compiler used within Software AG. In the future, support may be provided for the GNU and IBM C compilers.

Once the environment is supported, SMARTS provides a run-time library that is invoked by using the C headers supplied on the SMARTS source library. This causes the C compiler to call an external SMARTS API function stub, which passes control to the SMARTS nucleus to perform the requested function.

While the stubs themselves are independent of the environment, environment-dependent wrappers are required for each supported SMARTS environment and programming language. While the C code must be compiled once to use the SMARTS C headers, the object must be compiled for each different environment to include the appropriate wrappers. To get a C program that runs in one SMARTS environment to run in a different C environment requires that you link the C object with the appropriate SMARTS environment-dependent code.

Assembler Language Programs

Assembler language programs are similar to C language programs in that they call an environment-dependent program with exactly the same interface. The program may load the appropriate interface into a field that can be addressed by the interface, or the Assembler program may be linked with the appropriate environment-dependent Assembler interface in a way similar to C programs.

Other Programming Languages

Other programming languages use standard SMARTS high-level programming language interfaces that are known by the same name in all SMARTS environments. The environment dependencies are handled during the SMARTS environment installation and configuration stages so that programs written in any of these programming languages will run without change in the different SMARTS environments.

C-based Programs

Generation

Due to the nature of C programs and open-systems development techniques, there is a trend towards using "cross-compilers" on the open-systems platforms themselves or on PCs. These cross-compilers take C code and headers on the platform where they are running and produce system 370 object code that can be transferred to the S/390 mainframe system and linked to run on those systems. This solves problems such as the following:

- C supports external names longer than 8 characters, which is the standard on most S/390 mainframe systems. By using the file system on an open-systems or PC platform, there is no requirement to reduce external names longer than 8 characters to 8 characters and to make the appropriate changes for included header files.
- Most product generation uses makefile support, which does not port to the S/390 mainframe platforms.

The general goal is to compile and partially link code on the PC or open-systems platform and FTP the generated code to the target platform for a final link of the system and testing.

Currently, Software AG supports the System/C cross-compilers that can produce system 370 code. In the future, support may be extended to the GNU cross-compiler.

Compiling

During compilation, the cross-compiler uses the SMARTS C header files to resolve the external library names to the external SMARTS names.

Linking/Binding

After the object code has been created by the compiler, a two-phase link process is generally required.

The first phase, sometimes known as the "prelink", is run on the PC. Its purpose is to produce ESD format object modules that can be processed by the linkers /binders on all the target platforms. The prelink should also include the SMARTS independent stubs from the SMARTS SDK so that the only unresolved references are for the environment-dependent code that will be resolved during the final link.

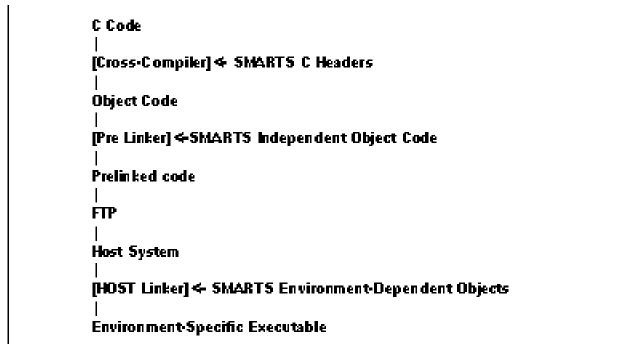
After all the prelinking, the second or final phase of the link occurs on the target platform. It involves including the platform SMARTS environment-dependent code required to run the code in a specific SMARTS environment.

Testing

Testing generally follows the compile and prelink phase on the PC or open systems platform. The prelinked code is FTPed to the target platform and relinked for the appropriate environment.

Testing will generally start in a batch environment. If the software is targeted exclusively at client environments, following successful testing on the initial platform the code can be transferred and linked for testing in the other target platforms and environments. If the software is destined for the SMARTS server environment, it must be linked for the SMARTS server environment and tested. Following successful testing on the initial platform the code can be transferred and linked for testing in the other target platforms.

Overview of C Program Generation for SMARTS



SMARTS C Program Generation Overview

Installation and Maintenance

The SMARTS server installation is a straightforward process of moving the SMARTS installation libraries onto the local system and running a number of simple jobs steps. The Software AG product that uses SMARTS is installed after SMARTS itself. The whole installation can then be verified using special procedures provided with the SMARTS-based software you are installing.

Once SMARTS is installed, multiple SMARTS-based Software AG products can be subsequently installed, with minimal configuration required.