Global Development System Manual Version V8.1 (draft copy)

Dec 2000

THE 32-BIT RUN-TIME ENVIRONMENT AND DEVELOPMENT SYSTEM

The first section of this appendix describes the changes to the GSM/Speedbase run-time environment that have been implemented for GSM V8.1j and V8.1k to support the execution of 32-bit applications. The second section provides an overview of the 32-bit Development System. The remaining sections describe some technical issues in full detail.

Important Note: No Production Versions of GSM V8.1j were released. Prior to external release, GSM V8.1j was superceded immediately by GSM V8.1k (see section 2.9). Although strictly, this appendix applies to both GSM V8.1j and GSM V8.1k, only the externally released V8.1k, which is a complete super-set of the "internal" V8.1j, is documented.

F.1 The 32-bit Run Time Environment

The following changes to the GSM/Speedbase run-time environment have been implemented to allow the execution of 32-bit applications.

- The Global System Manager start-up components have been modified to establish a number of data tables required for 32-bit application execution. These components, which are fully compatible with 16-bit GSM, are distributed on, and installed from, all V8.1k, and later, BACRES volumes;

- The Global System Manager program loader has been modified to recognise a 32-bit application and invoke the 32-bit loader/resolver to load the application, and any associated modules (see below) into dynamically allocated Memory Pages. These components, which are fully compatible with 16-bit GSM, are distributed on, and installed from, all V8.1k, and later, BACRES volumes;

- The 16-bit Debugging System has been supplemented by a simple, base-line 32-bit debugger ($DBG32 – see section F.3.7). These components, which are only required to debug 32-bit applications, are distributed on, and installed from, all V8.1k, and later, BACRES volumes;

- A number of new, 32-bit utilities are distributed on, and installed from, all V8.1k, and later, EPA/CFA volumes:

$BA32	32-bit frame loader (i.e. the 32-bit equivalent of 16-bit $BA);
$MH32	32-bit adjunct to the 16-bit $MH to load 32-bit frames for type "E" and "F" menu lines;
$MENU32	32-bit Menu Handler;
$MN32	32-bit Menu Maintenance (and various $MN*nnn* dependent frames);
$DLMMAIN	$$DLM index file maintenance utility (see section F.3.3.2);
$PAGES	Diagnostic utility to display the 32-bit page usage.

- A number of new, 32-bit modules are distributed on all V8.1k, and later, EPA/CFA volumes. These modules are only installed on the SYSRES volume for GSM (Windows) and GSM

(Unix) configurations:

$$DLM	Master DLM Index File (see section F.3.3);
P.$SDLM0	Library of system Dynamic Load Modules (see section F.3.3.2);
P.$MNMAP	Library of Dynamic Load Modules required by $MN32.

Thus, the 32-bit run-time environment is fully incorporated into GSM V8.1k. No extra software modules are required. The only action required to realise the benefit of the 32-bit run-time environment is to upgrade to GSM PM V8.1k.

Important Note-1: The 32-bit run-time option is supported on BOTH GSM-PM configurations AND GSM non-PM configurations. In general, for GSM-PM configurations the 32-bit components are distributed on, and installed from, the EPA volume. In general, for non-PM configurations, the 32-bit components are distributed on, and installed from, the CFA volume.

The Global System Manager (Windows) Global Client modules (i.e. GLOBAL.EXE and GLCONS.EXE) have been enhanced to include a dual 16-bit/32-bit Commercial Code Interpreter (CCI) and Resolve/Relocate SVC. These changes have been implemented in the V2.8 Global Clients. Consequently, in order to execute a 32-bit application on GSM (Windows) the BACNAT variant must be V2.8, or later;

Important Note-2: See the comment in section 2.8 regarding the availability of the latest GSM (Windows) BACNAT software from the Global web-site.

The Global System Manager (Unix) nucleus software (i.e. glintd) has been enhanced to include a dual 16-bit/32-bit Commercial Code Interpreter (CCI) and Resolve/Relocate SVC (SVC-79). The BACNAT variant of the first externally released 8.1k complaint 32-bit GSM (Unix) nucleus is V3.275.

The Global System Manager (Novell) nucleus software has NOT been enhanced to include a dual 16-bit/32-bit Commercial Code Interpreter (CCI) and Resolve/Relocate SVC (SVC-79). Consequently, it is NOT possible to execute 32-bit applications on GSM (Novell) configurations.

The Global System Manager (MS-DOS/Windows) nucleus software has NOT been enhanced to include a dual 16-bit/32-bit Commercial Code Interpreter (CCI) and Resolve/Relocate SVC. Consequently, it is NOT possible to execute 32-bit applications on GSM (MS-DOS/Windows) configurations.

The Global System Manager (BOS) nucleus software has NOT been enhanced to include a dual 16-bit/32-bit Commercial Code Interpreter (CCI) and Resolve/Relocate SVC. Consequently, it is NOT possible to execute 32-bit applications on GSM (BOS) configurations.

All the dual 16-bit/32-bit run-time components are switched into 32-bit mode by the presence of the Resolve/Relocate SVC in the GSM (Windows) or GSM (Unix) nucleus. Thus, the nucleus components effectively switch on the 32-bit *capability* (note that the 32-bit capability cannot be *realised* unless the options described below are enabled).

Important Note-3: At the time of writing 16-bit System Requests are not supported when running a 32-bit program. You are advised to use the $CUS Configuration Maintenance option to disable System Requests until this outstanding issue has been resolved.

Important Note-4: At the time of writing the Record/Playback option is not supported with 32-bit programs.

F.1.1 Customising the 32-bit run-time option on GSM (Windows)

The 32-bit run-time option for GSM (Windows) can be customised by the following options in the registry. All the ValueNames are under the following registry key:

..\Global\Client\Nucleus

Enable32Bit — This option must be set to “On” to enable the capabilities of the 32-bit run-time nucleus. The default setting is “On”;

PageTableEntries — This entry MUST be set to a non-zero value to enable the 32-bit run-time option in the GSM (Windows) Global Client. This option must be set to at least:

$$3 + (P * 6) + (Q * N)$$

where:

- *P* Number of Users (i.e. partitions) configured
- *Q* Average number of users executing 32-bit applications
- *N* Average number of pages required by the executing 32-bit applications

A "rule-of-thumb" value is 50 pages per user. All memory pages are allocated and de-allocated dynamically on a demand basis. The $PAGES utility, see above, can be used to obtain statistics from the Memory Page Allocator SVC;

TrapTableEntries — This value should not normally be required. The default of 50 should be adequate for most debug sessions;

LinkStackEntries — This value should not normally be required. The default of 50 should be adequate for most 32-bit applications;

SVC79Diagnostics — This option should only be enabled during problem investigations and should not be required during normal operation.

SVC79DiagLevel — This option should only be enabled during problem investigations and should not be required during normal operation.

F.1.2 Customising the 32-bit run-time option on GSM (Unix)

The 32-bit run-time option for GSM (Unix) can be customised by the following options in the Nucleus Options section of Configuration File:

32-bit option enabled — This option must be set to “Y” to enable the capabilities of the 32-bit run-time nucleus. The default setting is “Y”;

Number of 32-bit Pages tables — This entry MUST be set to a non-zero value to enable the 32-bit run-time option in the GSM (Unix) Global Client.

	See the comments above for the GSM (Windows) "PageTableEntries" registry setting;
Size of 32-bit Trap Table	This value should not normally be required. The default of 50 should be adequate for most debug sessions;
Size of 32-bit Link Stack	This value should not normally be required. The default of 50 should be adequate for most 32-bit applications;
32-bit diagnostics enabled	This option should only be enabled during problem investigations and should not be required during normal operation.
32-bit diagnostics level	This option should only be enabled during problem investigations and should not be required during normal operation.

F.2 The 32-bit Development System Overview

In order to avoid unnecessary complications with software ordering and installation the 32-bit Development System is a complete super-set of the 16-bit Speedbase Development. Thus the 32-bit Development System includes a number of modules (e.g. 16-bit sub-routine libraries such as C.$MCOB) that are not required for 32-bit program development!

The following new, 32-bit components are distributed on, and installed from, the UAA volume:

$SDL32	32-bit compiler;
P.$SDL32	32-bit compiler overlay library;
$SLDER32	32-bit compiler error messages file;
$MAP32	16-bit $FORM map to 32-bit DLM converter;
$FORM32	32-bit map DLM maintenance;
P.$FO32	DLM's required by $FORM32.

The remainder of this section describes the key features of the 32-bit Development System.

F.2.1 Memory Management

Addressable memory for each Frame or Program has been extended to approximately 64Kb (#0100 to #FF00). In addition the 32-bit language utilises a new and highly compact instruction set, which reduces the memory requirements of Procedural Statements by (typically) 30- 40%. The effective result of these changes is that 2 to 3 times more useable memory is now available for the operation of any single compiled program. For example, the 16-bit $PATCH utility occupies #8A9A bytes; the 32-bit equivalent (with some extra functionality) $PAT32, occupies #3F17 bytes.

F.2.2 Overlay Management

Each Speedbase Frame (or Dependent Frame) is now loaded into its own 64Kb page, and overlays therefore no longer impinge on the address space otherwise useable by controlling programs. The

number of 64Kb partitions available for the operation of an application is, in practice, now only limited by the amount of virtual memory available on the PC.

F.2.3 Dynamic Loading

System Service Routines are now dynamically loaded as and when needed, rather than being compiled into each application. This further reduces application memory requirements, and allows system services to be upgraded independently from application software.

F.2.4 Cobol Compatibility

A number of new facilities have been introduced within the 32-bit language to facilitate the migration of existing Global Cobol programs to the 32-bit Development System. These facilities include Common and External Areas, new Organisation Statements, Display Mapping, DMAM support, and a host of other minor enhancements. Except as further documented in this appendix, all Global Cobol Language statements supported by 16-bit $COBOL are supported by the 32-bit compiler, $SDL32.

The incorporation of these facilities into the 32-bit compiler, $SDL32, has allowed a number of enhancements to take place. Based items may now be coded anywhere within the Data Division, including the Common and External Sections. The GLOBAL statement has been similarly extended to allow based items to be declared as global. The FD statement has been extended to allow an explicit base pointer to be coded, and these based FD's may also be global items, and/or coded within Common and External Sections.

The operation of global symbols has also been enhanced. It is now no longer necessary to code a GLOBAL statement to permit access to an externally declared global symbol. Global data items may now be addressed directly from the referencing program without the need for any Data Division code.

F.3 32-bit Technical Information

The rest of this appendix describes the differences between the 16-bit Development Systems and the 32-bit Development System.

Important Note: Some external developers are using various copy-books from certain internal 16-bit copy-libraries. Although all externally documented copy-books are either compatible between 16-bit and 32-bit or, if incompatible, the differences are described below, **no attempt whatsoever has been made to maintain compatibly between internal 16-bit and 32-bit control blocks, data structures etc**. Any 32-bit developer using a copy-book that is not documented in a 16-bit Development Manual should log the problem with the Hotline.

F.3.1 Structural Differences between 16-bit and 32-bit applications

This section describes the Dynamic Linking process and associated program statements and structures used by 32-bit applications.

F.3.1.1 32-bit Program Structure Overview

Two types of programs can execute in the 32-bit environment: Load Modules and Executables. Load Modules contain systems routines and common services that are called by Executable programs. For example, the Speedbase Window Manager and Database Manager service routines have been written as Load Modules. The application itself is written using a number of Executables.

Both Load Modules and Application Frames are generated using the $SDL32 compiler.

Load Modules contain systems and service routines that may be used by the application software. When a 32-bit frame is compiled, these routines are not linked (or otherwise merged) into the application, but instead the compiler writes a table containing the names of the required Load Modules into the executable's. When the executable program is run, the Program Loader first examines this table, and loads any necessary Load-Modules into memory before execution commences.

When a program is compiled, the compiler first extracts all Load Module names and related Global symbols from the various Dynamic Load Module Libraries (e.g. P.$SDLM0 etc.). These routines and symbols are then automatically made available to the compilation. For system DLM's, this process is entirely automatic, and the application programmer does not need to specify any aspect of this process. For application DLM's, the DLM libraries must be specified using the $SDL32 LNK option (see section F.3.4.1.1).

The Dynamic Linking approach has a number of advantages over traditional linkage editing. Firstly, because no systems routines are linked into the executable program, the program is considerably smaller then the equivalent module, and therefore easier to distribute. Secondly, the Load Modules used by the application can be upgraded independently, without the needing to relink or recompile (or re-install) any application software.

F.3.1.2 Controlling and Dependent Frames

The 32-bit environment allows each frame to occupy a memory area of approximately 64Kb, which is roughly twice as much memory as was available in 16-bit Speedbase. Due to the compact nature of 32-bit code and the exclusion of system routine overhead, this is generally sufficient to accommodate a Frame containing more than 10,000 lines of procedure code. Since an executing program may be made up of any number of frames (each of which occupies it's own address space), the memory available to an application is in practice unlimited.

While the memory space available allows very large frames to be written, we strongly recommend that applications are segmented along functional lines into a number of smaller frames. This considerably improves application maintainability, and reduces overall memory requirement.

32-bit programs are segmented using the Controlling Frame / Dependent Frame mechanism which has been carried forward from 16-bit Speedbase. When a frame is declared to be a Controlling Frame, it may execute one or more Dependent Frames using the EXEC verb. The EXEC verb loads the specified Dependent Frame and passes control to it. The dependent frame may in turn also have dependent frames, allowing the construction of multi-level overlay structures.

The Dependent frame is loaded and executed using an EXEC statement within the controlling frame, and it is automatically unloaded when the dependent frame executes an EXIT statement from its highest level of control. If the Dependent Frame caused any Load Modules to be loaded, then these are also automatically unloaded at that time.

A Dependent Frame can generally access all symbols declared within its Controlling Frame. When a frame is designated to be a controlling frame, then all symbols defined within it are generally deemed to be Global. When the Dependent frame is compiled, these symbols are automatically made available to the compilation, just as if they had been coded within the dependent frame. The only coding requirement is to specify the name of the controlling frame within the FRAME statement, and thereafter the process is again entirely automatic.

The addresses of Global Symbols referenced by the dependent frame are resolved when the Dependent frame is loaded. It is therefore possible to modify a controlling frame without needing to recompile its dependents. Recompilation is only necessary if a change is made to an existing Global symbol's picture clause, or, if the item is indexed, its number of occurrences and group-length. This means that substantial modifications can be made to a frame without affecting related programs, and only amended frames need be distributed and installed in maintenance releases.

F.3.1.3 Common Service routines

An application will often contain a number of service routines that are called from different programs within the application, such as common routines that do complex calculations, or provide enquiry look-ups. While these common routines could be copied into each program, it is often more convenient (and elegant) to code them as separate service programs. Such programs can be coded as a service overlay, which may then be EXED'ed when needed, or may be coded as Application Load Modules.

When coded as a Load-Module, the routine will be loaded into memory when any executable that references it is itself loaded, and therefore remains memory resident for the duration. If coded as a service overlay, the routine is loaded (and unloaded) each time it is called. The main consideration here is therefore to determine how frequently a service will be called, and whether repeated loading would cause an unacceptable system overhead.

A service such as a frequently called calculation is generally best coded as a Load-Module. An infrequently called service such an enquiry window is probably best coded as a service overlay.

F.3.1.3.1 Application Service Overlays

A service overlay is coded like any normal executable program. However, unlike 16-bit Speedbase, an executable or dependent program may be EXED'd from anywhere within an overlay structure, so long as all global symbols referenced by it can be resolved when it is loaded. This means that a non-dependent program can be EXED'd by any executable, and a Dependent overlay may be EXED'd not only by it's controlling frame, but also by all other lower level overlays within the structure.

Parameters can be passed to a service overlay using the USING clause on the EXEC statement, which are subsequently received using a corresponding ENTRY USING statement in the service overlay. If the service overlay needs access to other data items in the calling program, then it must be coded as a DEPENDENT program. If this is done, then the service overlay will only be executable by its controlling frame, or overlays EXED'ed by that controlling frame.

F.3.1.3.2 Application Load Modules

Application Load Modules are also generated using the 32-bit compiler. The source program begins with a FRAME or PROGRAM Statement as usual, and additionally contains a LOAD-MODULE statement in the frame header section. Otherwise there are no special coding requirements. While all symbols defined in the Module may be treated as Global, it is more usual to allow global access to the few specific variables that are going to be referenced. This is achieved using the GLOBAL statement discussed later in this appendix.

When a Frame (or a Dependent Frame) is compiled that requires access to the Load Module, the compiler needs to be informed of the module's existence. This is done using Compiler option LNK, which requests the Load Module name and unit. This compiler option causes the Load Module's global symbols to become available to the compilation. If an application makes use of a number of

Load-Modules, it is often convenient to combine them in a program library, then only the library need be specified and all load modules within it will be available to the compilation. Note that the library may contain a mixture of Load-Modules and executables, as the compiler LNK option only extracts global symbols belonging to Load-Modules.

Section F.3.3.2 describes the rules for loading Dynamic Load Modules.

F.3.2 Overview of New Language Elements introduced into the 32-bit language

This section describes the language elements that relate to Speedbase overlay structures and Dynamic Linking. For all other language statements please refer to the V8.1 Speedbase Development Manual.

F.3.2.1 FRAME AND PROGRAM Statements

The first statement in each Speedbase program is either a Frame or Program statement. It is coded:

FRAME *name1* [DEPENDENT ON *name2*] [*"Program Title"*]

or:

PROGRAM *name1* [DEPENDENT ON *name2*] [*"Program Title"*]

Where *name1* is the name of the program being compiled, and, if coded, *name2* is the name of the program's Controlling Frame. The keywords FRAME and PROGRAM have identical function.

When the DEPENDENT ON clause is coded, this specifies that the program is dependent on the controlling program given by *name2*. During compilation, this option causes the Speedbase Compiler to extract the locally defined Global symbols from *name2*'s object file, which are then made available to the compilation. Note that if *name2* is itself Dependent on a further program (such as in a multi level overlay structure), then the compiler includes the global symbols of all the higher level program(s) as well.

The Dependent on Clause has two further effects. When a frame is specified to be a dependent frame, it cannot be executed directly from the GSM Menu (or the GSM Ready prompt). Instead it must be executed using the EXEC Verb. Secondly, when the Dependent frame is executed, the Speedbase Loader checks to ensure that it's Controlling Frame *name2* is resident in memory. If *name2* is not resident, then the loader will immediately terminate the frame.

Note that *name2* does **not** have to be a designated Controlling Frame, and unlike 16-bit Speedbase, there are no special requirements for the EXECing program. It is permissible for *name2* to be an Executable Program, a Dependant Program, or a DLM. Nor is it necessary for the dependant frame to be EXED'ed directly by *name2.* The only requirement is that *name2* must be resident in memory at the time that *name1* is EXED'ed. The identity of the EXEC'ing program is also not important, the Dependant Program can be EXED'ed from any program at any time, provided that *Name2* is resident at the time.

F.3.2.2 LOAD-MODULE Statement

This statement is coded in the Frame Header area and specifies that the program is to be compiled as a Dynamic Link Module. It is coded:

LOAD-MODULE

This statement must be coded for the program to be dynamically loadable.

F.3.2.3 CONTROLLING FRAME Statement

This statement tells the compiler that the Frame will execute Dependent Frames, and that all locally defined Symbols within the program are to be regarded as Global (see also the LOCAL SECTION). It is coded:

CONTROLLING FRAME

The statement is a compiler directive that causes all further data declarations and Procedure Division labels to be treated as Global Symbols. All further symbol declarations will generally be treated as Global except for those symbols defined within the Linkage Section, External and External Sections and the Local Section. Note that individual symbols within these sections may still be declared to be global by use of the GLOBAL statement.

Important Note: Unlike 16-bit Speedbase, it is **NOT** necessary for a Frame to be a designated Controlling Frame, in order for it to have associated Dependant Frames.

When the Controlling Frame Statement is NOT coded, then the symbols defined in the compilation are generally treated as local, and cannot therefore be addressed by external routines. The only exceptions to this are certain Symbols generated for Systems purposes, I/O channels, and symbols defined by the Entry Statement and Common Section Statements. These symbols are always treated as Global symbols irrespective of the chosen compiler options.

All symbols relating to Database I/O channels declared using the ACCESS statement are Global by default, irrespective of the preceding discussion. If a local I/O channel is required, then the new LOCAL clause must be coded in the Access statement as follows:

ACCESS [LOCAL] [*dbid:*] *rt1* [*rt2 ... rtn*]

When the LOCAL clause is coded, the I/O channel can only be accessed by the program in which it is declared. **To avoid global symbol name clashes, this option will usually be used for all I/O channels declared within Load-modules**.

F.3.2.4 GLOBAL Statement

This statement is used to specify that a locally defined Symbol is Global. It is coded:

GLOBAL *Name1* [*Name2*... *Name n*]

Where *Name1*, *Name2*, and *Name n* are the names of Symbols that are locally defined within the program.

There is no need to use this statement to declare that externally defined symbols are global. This is because all external globals are known at compilation time and are already known to be Global. Unlike the Cobol compiler, $COBOL, the statement will generate a warning if used to reference external Global Symbols.

The Symbol Names coded can reference any Locally Defined Symbol within the compilation, including all Data Items, FDs, Windows, PFs, and Entry-Points. Unlike the Cobol compiler, $COBOL, Based Items may be declared to be Global, provided that the Base Pointer Symbol of the Item is itself also a Locally Defined Global Symbol.

The Picture Clause and other information relating to each Global Symbol is passed on to any program that references it. An Externally Declared Global Symbol can therefore be treated by the referencing compilation just as if it had been locally declared. This allows a Global Data Item to be referenced directly by, say, the MOVE statement or any other Procedure Division, Window Division, or Data Division construct.

F.3.2.5 LOCAL SECTION

This new Data Division Section specifies that the symbols defined within it are local (i.e. non-Global symbols). It is coded:

```
LOCAL SECTION
```

The section is generally only used in combination with the Controlling Frame option, to specify that certain Data Division items should be local to the compilation. The Statement may also be used to terminate a Global Section. The option reduces the number of Global Symbols present, which may improve loading time for both the program itself, and any dependent frame it executes. Because unreferenced local variables are not included in the program's Symbol Table, this option may also reduce the total memory required by the program.

F.3.2.6 GLOBAL SECTION

This new Data Division Statement is used to declare that all Data Division symbols defined after the statement are to be treated as Global Symbols. It is coded:

```
GLOBAL SECTION
```

The statement provides a convenient way of declaring a large number of global items. It has the same effect as coding each individual symbol name after a GLOBAL statement (which can be somewhat tedious).

F.3.2.7 COMMON SECTION

This section is used to declare a common Data Division area which may be addressed by external Modules. It is coded:

```
COMMON SECTION Name
```

Where *Name* is a Symbol name. The statement is followed by a number of data division items, and ends at the beginning of the next Section or Division. The Statement causes the compiler to generate a Global Data Item *Name*, which contains the start address of the section and total length of the section. The items within the section may then be addressed by use of the External Section Statement (see below).

It is important to note that statement simply declares an area of memory which may be addressed externally, however the individual symbols coded within the section are treated as local, unless explicitly specified as Global using the GLOBAL statement. The Section may contain any Data Division Items and constructs whatsoever, including BASED items.

When Based Items are coded within the Section, the base pointer of the item does not necessarily need to be coded within the same Section. Equally when the Section includes Redefinition's, the Redefined item may be a Symbol declared outside of the common section. However, if either of

these options are made use of, care must be taken that the referenced symbol is either a Global Symbol or a symbol that is declared within a further Common Section, so as to be available to any referencing program.

F.3.2.8 EXTERNAL SECTION

This section is used to address a Common Section area of memory that resides in an external program. The statement is coded:

EXTERNAL SECTION *Name*

Where *Name* is the symbol name previously coded within the Common Section statement of the external program. In order to address the external section, it must either reside in one of the program's Controlling Frames, a Dependent Module or within a Load Module that was specified to the compiler using the LNK compiler option.

The statement **MUST** be followed by the same sequence of Data Division statements as declared following the Common Section statement. This is most easily achieved by copying the Data Division items from a previously established copybook using the COPY statement.

When a program containing an External Section is compiled, the length of the two sections are compared to ensure that they are equal. The compiler will generate a warning if the External Section is shorter than the Common Section. If the External Section is longer than the Common Section, then the compiler will generate an error, and the program will not be executable. Note that it is critical that the two Data Division segments are identical as any differences will result in a Resolver Error at program load time.

F.3.2.9 Programming Notes

Application programs will normally only make use of the Global Section, Controlling Frame and Local Section Statements to facilitate inter-program data access. The Common/ External Section facility has been provided to allow compatibility with Cobol Programs, and will generally only be used within Systems Routines.

The only advantage of the Common section is that it allows a large number of variables to be addressed while using only a single Global Symbol. The variable names within Systems data structures therefore remain local, which avoids inadvertent references and variable name clashes with application programs.

F.3.3 General Considerations for 32-bit Developers

This section describes some general considerations that all 32-bit developers should be aware of.

F.3.3.1 Load Module and Global Symbol Naming Conventions

The various Load Modules available to the 32-bit loader can be divided into 3 types:

- System Dynamic Load Modules (DLM's), provided with Global System Manager. These DLM's are held in the System DLM's libraries P.$SDLM0 - P.$SDLM9;

- Utility Dynamic Load Modules (DLM's) also provided with Global System Manager that provide the functionality required by 32-bit run-time and development utilities (e.g. FO$RTN in P.$FO32 for use by $FORM32);

- Application Dynamic Loaded modules, provided with Global 3000 or developed by resellers using the 32-bit Development System.

The name of every global symbol defined in a System Load Module will include the "$" character. Thus, to avoid symbol name clashes, **THE "$" CHARACTER SHOULD BE CONSIDERED A RESERVED CHARACTER AND MUST NOT BE INCLUDED IN THE NAME OF GLOBAL SYMBOLS IN APPLICATION DYNAMIC LOAD MODULES**.

The only exception to the "$ rule" is the BREXEC entry-point in the BR$EXE DLM, within the P.$SDLM0 library. The BREXEC entry-point has been preserved for compatibility with the 16-bit Global Reporter interface.

F.3.3.2 Load Module Loading Rules

With the exception of the main System DLM library, P.$SDLM0, the names of the various DLM libraries available to the 32-bit compiler must be specified, at *compile-time*, explicitly using the LNK option (see section F.3.4.1.1).

In addition to the *compile-time* DLM access, the 32-bit Frame/Loader/Resolver must search the various DLM libraries to resolve external symbols at *load-time*. The Loader uses the following search algorithm when loading both System and Application DLM:

Library	*Unit*	*Via Index File*	*Index File unit*
P.$SDLM0	$$D	none	none
P.$SDLM[1-9]	$$D	none	none
variable	variable	$$DLM	$$D
variable	variable	$$DLM0	$P
variable	variable	$$DLM1	$P

The P.$SDLM0 to P.$SDLM9 DLM libraries are standard (100-entry) program libraries. The new $$D unit assignment is set to SYSRES when GSM is installed. These libraries are reserved for System DLM's.

The $$DLM, $$DLM0 and $$DLM1 are simple index files, created and maintained, using the $DLMMAIN utility. Each DLM Index File holds a list of up to 80 DLM libraries and units.

The $$DLM Index File is reserved for use for Utility DLM's. The default $$DLM Index File installed on $$D contains the following entry:

P.$MNMAP	$$D

When the 32-bit Development System is installed, the $$DLM library is updated to include the following extra entry:

P.$FO32	SYSSBD unit

The $$DLM0 DLM Index File is reserved for use by end-user 32-applications. The $$DLM1 DLM Index File is reserved for use by Global 3000. Note that the DLM search order allows end-user DLM libraries to replace standard, Global 3000 libraries.

The $DLMMAIN utility is available to create and maintain the DLM Index files. Furthermore, the

DLMM$ routine is available to access these Index files under application program control.

F.3.3.3 32-bit System Variables, Sub-routines and DLM's

The following 16-bit compatible System Variables, as documented in the various V8.1 Global Cobol development manuals, are available to 32-bit programs:

$$ARCH	$$AREA	$$ATIM	$$AUTO	$$BELL	$$CLR
$$CODE	$$COND	$$CRES	$$DATE	$$DECP	$$DOWK
$$EOF	$$ESC		$$FED	$$FESC	$$FLSH
$$GUI					
$$GWW	$$HBK	$$HCLR	$$HFIL	$$HPTR	$$HUN
$$INDE	$$INT	$$JCL	$$L	$$LENG	$$LEVN
$$LIB	$$LINE	$$LNID	$$MACH	$$MCOD	$$MNID
$$MU	$$NCYR	$$PAGE	$$PAR	$$PGM	$$PM
$$PRIN	$$PROM	$$PVOL	$$REV	$$REL	$$RES
$$RUN	$$RWID	$$SEED	$$SER	$$SNAM	$$SRKT
$$SVSR	$$SYSM	$$TER	$$TERM	$$TYPE	$$ULEV
$$USA	$$VERS	$$WIDE	$$X	$$XY	$$Y
$$ZERO	$$AUTH	$$OPID	$$SCNN	$$SUSP	$$TASK
$$USER					

The following new 32-bit System Variable are also available to 32-bit programs:

$$EPT	PIC PTR	32-bit program Entry-Point
$$EOJ	PIC PTR	32-bit End-of-Job routine
$$AR32	PIC X(256)	Extended $$AREA for 32-bit programs
$$3000	PIC X(1024)	Extended work area for Global 3000

The following 16-bit subroutines have been converted into 32-bit and are available in various DLM's in the P.$SDLM0 library:

Sub-routine	*DLM*	*Notes*
BDAM Access Method	BO$AMS	
DLAM Access Method	BO$AMS	
DMAM Access Method	BO$DMS	
DVAM Access Method	BO$AMS	
ISAM Access Method	BO$AMS	
Open Direct AM (Unix)	BO$AMS	
Open Direct AM (Win NT)	BO$AMS	
Open TFAM	BO$AMS	
PSAM Access method	BO$AMS	
RSAM Access Method	BO$AMS	
TFAM Access Method	BO$AMS	
VLAM Access Method	BO$AMS	
ACCE$	BO$CON	
AS-EB$	BO$CNV	
AS-RL$	BO$CNV	
ASSIG$	BO$FIL	
BASEL$	BO$EDI	
BASEX$	BO$EDI	
BI-BS$	BO$CNV	

BI-HX$		BO$CNV	
BI-OC$		BO$CNV	
BOX$		BO$EDI	
BOXU$		BO$EDI	
BREXEC		BR$EXE	
BS-BI$		BO$CNV	
CALC$		BO$FIL	
CATA$		BO$FIL	
CEOL$		BO$CON	
CEOS$		BO$CON	
CHAR$		BO$CON	
CHARX$		BO$CON	
CHECK$		BO$CON	
CHKPR$		BO$CON	
CID-D$		BO$SMS	
CLEAR	verb	BO$CON	
CLEAR$		BO$CON	
CLOSE$		BO$FIL	
COLOR$		BO$CON	
CONV$		BO$FIL	
COPY$		BO$FIL	
COPYP$		BO$FIL	
DBRPK$		BO$DMS	
DBCLC$		BO$DMS	
DBKES$		BO$DMS	
DBREK$		BO$DMS	
DBRFK$		BO$DMS	
DBRLK$		BO$DMS	
DBRNK$		BO$DMS	
DBRTT$		BO$DMS	
DBSTA$		BO$DMS	
DBUF$		BO$CON	
D-CID$		BO$SMS	
DELE$		BO$FIL	
DENAT$		BO$EXT	
DEVIN$		BO$FIL	
DISP$		BO$CON	
DIVID$		BO$CNV	
DL-DT$		BO$DAT	
DLMM$		BO$FI2	See note 2
DOWK$		BO$DAT	
DS-DT$		BO$DAT	
DSYSR$		BO$CON	
DTCLS$		BO$DMS	
DT-DL$		BO$DAT	
DT-DS$		BO$DAT	
DT-DY$		BO$DAT	
DTOPN$		BO$DMS	
DTWRT$		BO$DMS	
DY-DT$		BO$DAT	
EB-AS$		BO$CNV	
ECHO$		BO$CON	
EDIT verb		BO$CON	

ELIST$	BO$FIL	
EOFCH$	BO$CON	
EOJ$	BO$JOB	
ESYSR$	BO$CON	
EXIT$	Not implemented for 32-bit	
FDAT$	BO$DAT	
FILDF$	BO$FIL	
FILE$	BO$FIL	
FINFO$	BO$FI2	
FIX$	BO$FIL	
FMODE$	BO$CON	
FNDME$	BO$PSS	
FREEX$ (was FREE$)	BO$SMS	See note 1
FSTAT$	BO$FIL	
GETX$	BO$SMS	
GETXN$	BO$PSS	
GTDES$	BO$FIL	
GTSCR$	BO$CON	
HMS-T$	BO$DAT	
HX-BI$	BO$CNV	
IRBLD$	BO$DMS	
ISUSE$	BO$FIL	
JOB$	BO$JOB	
KCR$	BO$EDI	
KCRX$	BO$EDI	
KEYS$	BO$CON	
LIBR$	BO$FIL	
LIBRA$	BO$FIL	
LINE$	BO$EDI	
LINFO$	BO$FI2	
LIST$	BO$FIL	
LOCK REGION verb	BO$FIL	
LOG$	BO$SMS	
LOGOF$	BO$SMS	
LWORK$	BO$FIL	
MENU$	BO$EDI	
MIDCH$	BO$DAT	
MIDN$	BO$DAT	
MSG$	BO$CON	
MULTI$	BO$CNV	
NAR$	BO$CON	
NCLOS$	BO$EXT	
NKM-C$	BO$CNV	
NLIST$	BO$EXT	
NLOGF$	BO$SMS	
NOPEN$	BO$EXT	
N-ORG$	BO$PSS	
NPART$	BO$CON	
NSCR$	BO$CON	
OC-BI$	BO$CNV	
OPDE$	BO$FIL	
OPEN$	BO$FIL	
OPENS$	BO$FIL	

OPID$	BO$SMS
OPIDX$	BO$SMS
OPNM$	BO$SMS
ORG-N$	BO$PSS
PAGE$	BO$FI2
PASS$	BO$CON
POSI$	BO$CON
POSIC$	BO$CON
PRIN$	BO$CNV
PROG$	BO$PRG
PTDES$	BO$FIL
PTSCR$	BO$CON
PWCHK$	BO$CNV
PWNUL$	BO$CNV
PWNUM$	BO$CNV
QINDX$	BO$PRG
QLOAD$	BO$PRG
QSRT$	BO$SRT
RAND$	BO$CNV
RDIMG$	BO$CON
RDSCR$	BO$CON
RELX$	BO$SMS
RENA$	BO$FIL
RENAT$	BO$EXT
RENAX$	BO$FI2
REST$	BO$DSS
RL-AS$	BO$CNV
RXLAT$	BO$CON
SAVE$	BO$DSS
SAVEN$	BO$DSS
SBTYP$	BO$FIL
SCAN$	BO$FIL
SCHEM$	BO$EXT
SCOLR$	BO$CON
SCR$	BO$FIL
Screen formatting	BO$FOR
SCRN$	BO$CON
SCROLL Verb	BO$CON
SECS$	BO$DAT
SECUR$	BO$FIL
SET$	BO$FIL
SHCMD$	BO$EXT
SLOCK$	BO$FIL
SORT verb (MSORT$)	BO$SRT
SQRT$	BO$CNV
SUBS$	BO$FIL
SULOC$	BO$FIL
TEST$	BO$CON
TEXT$	BO$CNV
TFSCM$	BO$EXT
T-HM$ (sic)	BO$DAT
T-HMS$	BO$DAT
TIME$	BO$DAT

TIME2$	BO$DAT
TSRT$	BO$SRT
TXEDT$	BO$EDI
TXVAL$	BO$EDI
TYP$	BO$CON
TYPF$	BO$CON
TYPW$	BO$CON
UAD$	BO$FIL
USER$	BO$SMS
USERX$	BO$SMS
USTAT$	BO$FI2
UWORK$	BO$FIL
VIDEO$	BO$CON
VOLID$	BO$FIL
WIDE$	BO$CON
WINDO$	BO$FOR
WOYR$	BO$DAT
WRES$	BO$FOR
WXLAT$	BO$CON
ZERO$	BO$CNV

The following new 32-bit subroutines have no 16-bit equivalents:

Sub-routine	*DLM*	
CAL16$	BR$EXE	See note 5
CPTR$	BO$PSS	See note 3
ELOCK$	BO$FIL	See note 4
FREEX$	BO$SMS	See note 1
MH$	BO$MEN	

Note-1: The 16-bit FREE$ sub-routine has been replaced by the 32-bit FREEX$. The calling interface for FREEX$ is as follows:

CALL FREEX$ USING *fm*

where *fm* is the Free Space Management area:

```
01   FM
  02 FMFUN     PIC 9 COMP        * Function required
                                 * 0 = Get work space
                                 * 1 = Free work space
  02 FMSIZE    PIC 9(6) COMP     * Size of work space required
  02 FMPTR     PIC 9(9) COMP     * Pointer to the 1st byte allocated
```

Note-2: The DLMM$ routine can be used to maintain DLM Index Files. The calling interface for DLMM$ is as follows:

CALL DLMM$ USING *fd table flag*

where: *fd* closed FD containing the name and unit of the DLM Index File

table control table of the following format:

```
01    TB
 02   TBENTRY   PIC 9(4) COMP          * Number of entries
 02   TBE OCCURS 80
  03  TBLIB     PIC X(8)               * DLM library name
  03  TBUID     PIC X(3)               * DLM unit number
```

flag PIC 9(4) COMP variable or literal:

0 return the contents of the DLM table
1 write back table (creating new DLM Index File if necessary)

Note 3: The CPTR$ routine adds an offset to a 32-bit pointer. The offset can be either a positive or a negative quantity. The calling interface for CPTR$ is as follows:

CALL CPTR$ USING *ptr1 offset ptr2*

where: *ptr1* PIC PTR source pointer (unchanged by the routine)
offset PIC 9(4) COMP offset value
ptr2 PIC PTR destination pointer

Note 4: The CAL16$ routine replaces the EXEC statement as the technique used to invoke a 16-bit program from a 32-bit application. For example, if an application needs to invoke a 16-bit job, MYJOB, replace the 16-bit:

EXEC “MYJOB”

by the 32-bit equivalent:

CALL CAL16$ USING “MYJOB “

Note the extra spaces in the quoted string to pass the single parameter as a PIC X(8) quantity. Note also that control will always return back to the 32-bit application unless the 16-bit program/job terminates with a Program Check.

Note-5: The calling interface to the 32-bit Menu Handler is as follows:

```
MOVE 0 to MIPTR          * i.e. redefined as PIC 9(9) C
CALL MH$ USING MIPTR
* set up rest of MI-block
CALL MH$ USING MIPTR     * display menu and wait for option keyed
```

F.3.3.4 32-bit Speedbase System Variables and Middleware

A number of extensions to the Speedbase run-time that were previously only available in the 16-bit Global 3000 Middleware have been incorporated into 32-bit Speedbase (i.e. have become the 32-bit Speedbase Middleware). The new functions are included in the BA$MID, BA$POP and BA$QFT DLM’s within the P.$SDLM0 library.

Important Note: The 32-bit Middleware also contains a number of additional DLM libraries (e.g. P.MIDDLE, P.SEARCH, P.COMMON etc.). Details of the contents of the Global 3000 Middleware DLM’s are beyond the scope of this document. However, developers should be aware that a strict naming convention for Entry-Points and other global symbols in the Global 3000 Middleware does

not exist (cf. the "$" rule for all system DLM's in the P.$SDLM0 library). This issue is under review.

F.3.3.4.1 32-bit Speedbase System Variables

The following 16-bit compatible Speedbase System Variables, as documented in the V8.1 Speedbase Development Manual, are available to 32-bit programs:

$BKFR	PIC X(6)	Back frame to load id EXEC'ed
$BLST	PIC S9(2)C	Baseline Status variable
$FNBY	OCCURS 18 PIC X	Hex codes for functions 1-18
$FNBY0	PIC X	Hex code for function-0 <CR>
$FNTX	OCCURS 18 PIC X(7)	Keytop names for functions 1-18
$FNTX0	PIC X(7)	Keytop name for function-0 <CR>
$FUNC	PIC 9(2) C	Function number returned
$FWFR	PIC X(6)	Forward frame to load if OK
$INCD	PIC X(32)	Common data area
$INPC	PIC X(2)	Product code
$LINO	PIC 9(4) C	Current printer line number
$MODE	PIC 9 C	Current screen mode
$PGLN	PIC 9(4) C	Current printer page length
$PHLT	PIC 9 C	Disable printer halt switch
$PGNO	PIC 9(4) C	Current printer page number
$PRUN	PIC X(3)	Printer assigned unit
$RSPG	PIC 9(4) C	Page number to restart from
$P-IM	PIC PTR	Pointer to IN-block
B$P-FT	PIC PTR	Pointer to Function Key Hlp text block

F.3.3.4.2 32-bit Speedbase "Legacy" Middleware

The following 32-bit Middleware functions have been converted from equivalent 16-bit functions:

Entry	Parameters	Description
B$BLEN	*length*	Set input field length
B$SLEN	*length* *shift*	Set input field length and shift length
B$SETA	*N*	Set field attribute value to N
B$NSTX	<none>	Set field attribute to 4 (non-scrolled text)
B$STXT	<none>	Set field attribute to 3 (scrolled text)
B$BXCL	<none>	Set field attribute to 9 (boxes and lines)
B$HILI	<none>	Set field attribute to 8 (base-line errors)
B$DPLN	*wy* *length*	Change dependency length
B$DNTC	*pointer*	Internal routine for Global 3000 only

B$ECHO	<none>	Set field change flag to force echo
B$STLN	*wy* *field* *length*	Change length of field in window
B$DLRN	*rc*	Return the current record number of I/O channel
B$DRGN	*rc*	Return the current generation number of I/O channel
B$INIR	*rc*	Initialise I/O channel record
B$CURR	*wy*	Check current record status
B$LASR	*wy*	Check if cursor at bottom of the screen
B$FFUL	*vi*	Return information on the last I/O channel
B$SPMD	*wy* *modes*	Enable/disable permitted modes in specified window
B$TTLS	*window-id*	Set title line on window
B$QFT	*window-id* *forward-routine* *backward-routine*	This routine intercepts database I/O operations and allows the calling program to detect when an invalid record is found. The calling program can perform a direct fetch rather than reading all the intervening records suppressing them from the display, which can be slow. The *forward-routine* and *backward-routine* routines are called when an I/O error has occurred to check the returned record and replace and/or signal the end of the file. They are called for forward reads (FETCH NEXT) and backward read (FETCH PRIOR).
B$POP	*window-id* [*clear-flag*]	Pop Menu handler with optional clear flag.
B$POPS	*window-id* *list* [*char*]	This routine changes the first character in the pop-menu lines to the passed character, where *window-id* is the Window ID of a pop-menu, *list* is a list of section letters to alter terminated by the character “z”, and *char* contains the character to which the 1st character of each menu line will be changed. If char is not given a default of “*” is used.
B$CLHI	<none>	This routine redisplays the highlight menu line in the Window Background attribute.

F.3.3.4.2 32-bit Speedbase “New” Middleware

In addition to the familiar Middleware functions listed above, a number of new functions and System variables are now available:

F.3.3.4.2.1 B$GTWK

The B$GTWK function returns a “sanitised” version of the Window Control Block structure.

F.3.3.4.2.2 B$GTIO

The B$GTIO function returns a “sanitised” version of the I/O control block.

F.3.3.4.2.3 B$ST2N

A new entry-point, B$ST2N, has been added to the Database Status Routine BA$STA. A Call on BA$ST2N permits the Database name to be passed (as opposed to a Record Control Block) as follows:

```
CALL B$ST2N USING Dbname S2
```

Where Dbname is a X(8) database name (including the "DB" prefix), and S2 is the long version of the parameter block as otherwise normally returned by an equivalent call to B$ST2.

Note that the routine requires the Database name (i.e. the file-name of the Main Index File), to be supplied. Note that this is different from the Database ID.

F.3.3.4.2.4 B$CNV

A new entry-point into the Window Manager, B$CNV call has been implemented which converts a field to upper-case using the following call:

```
CALL B$CNV USING Window Field
```

where Window is a window-ID, and Field is the name of a field that has been coded within the window.

The call must be made **before** the window is entered or displayed. After the call, the field will be converted only as part of the Accept operation. Note that if the field is not accepted, then no conversion will occur

F.3.3.4.2.5 $BLST

A new System Variable $BLST has been added, which contains the Base-line status. This variable should not be modified.

F.3.3.4.2.6 B$FRER

A new entry-point, B$FRER, has been implemented which returns the # of free records for a given record type. The call is of the form:

```
CALL B$FRER USING DBNAME RECID FREE
```

Where DBNAME is a PIC X(8) database name (e.g. "DBDEMON"), RECID is a PIC X(2) Record-ID (e.g. "RT"), and RECID is a 9(9) Comp variable into which the number of free record slots are returned.

The database must be opened before calling B$FRER, and Exception code 25550 will be returned if the database is NOT open, or if the routine is unable to open the Database Dictionary.

The number of free records, as returned in FREE, gives the number of empty record slots in a GSM Native Database. For other database types, the number returned = 99,999,999 less the number of records in the file.

F.3.3.4.2.7 $DBXP

A new Database Intercept facility has been provided using the new system variable $DBXP as demonstrated in the following Procedure Division extract:

```
POINT $DBXP at Etp-name

ENTRY Etp-name USING X1 R1
```

The Point statement points system variable $DBXP at the application entry-point Etp-name. Thereafter Etp-name will be called by the Database manager immediately prior to the commencement of processing of the Database Operation.

The Entry statement declares the Intercept routine Etp-name, and must declare X1 and R1 as parameters. Parameter X1 contains details of the requested operation (Opcode, Index etc), and R1 contains a "sanitised" version of the I/O channel's Record control block (RCB).

The intercept routine will normally return control using an EXIT statement, which causes the DBMS to complete processing of the request. If an exit condition is signalled, then the DBMS will suppress the operation, and return the signalled exception number to the statement that called the DBMS. Note that exceptions should not be passed back where the I/O was performed under the control of the Window Manager, since it will be unable to process the exception correctly.

Note also that $DBXP must be reset to HIGH-VALUES as soon as the intercept is no longer required, and failure to do so will result in memory Violations, or other unpredictable errors. If the intercept routine itself needs to perform I/O operations, then it should first reset $DBXP to high-values (or be coded to handle the subsequent call to the intercept routine).

F.3.4 Migrating 16-bit Speedbase Applications to 32-bit

This section describes the factors that must be considered when converting 16-bit Speedbase applications to 32-bit.

Important note 1: Developers should read section F.4 which describes some important factors that must be considered in order to get the best performance, both at compile-time and at run-time, when developing large applications.

Important note 2: 32-bit Speedbase frames that don't require a pre-opened database can be loaded directly from the menu prompt or the command prompt**. DO NOT ATTEMPT TO USE $BA TO LOAD A 32-BIT FRAME**. If it is necessary to provide a 32-bit frame with an open Speedbase database the 32-bit equivalent of $BA (i.e. $BA32) can be used to perform the open and pass control to the 32-bit frame.

In general the 32-bit compiler, $SDL32 is a complete superset of the 16-bit compiler, $SDL. All the language statements provided in $SDL are supported in $SDL32. Most 16-bit Speedbase programs can be compiled directly using $SDL32 without modification of any kind. There are, however, a number of minor differences between $SDL and $SDL32 which should be considered when migrating an application.

F.3.4.1 Compiler Dialogue Changes

This section describes the differences between $SDL and $SDL32. The $SDL32 dialogue is extremely close to, and deliberately based on, the $SDL dialogue.

The most striking difference between $SDL and $SDL32 is that $SDL32 is that although the actual linking takes place at program load/resolve time – the compiler must still be aware of the external dependencies. This difference in functionality leads to a slightly different dialogue. The following example dialogue for $SDL:

```
GSM P1 READY:$SDL
$SDL V8.1
$A3 SOURCE:TEST16 UNIT:210
$A3 OBJECT  UNIT:210  SIZE:64000
$A3 DICTIONARY 1:TEST1 UNIT:210  GENERATION 37
$A3 DICTIONARY 2:<CR>
$A3 LISTING UNIT:202 SIZE:<CR>
$A3 COMPILATION OPTION:COP
$A3 COPY LIBRARY:COPY1  UNIT:211
$A3 COPY LIBRARY:COPY2  UNIT:211
$A3 COPY LIBRARY:<CR>
$A3 COMPILATION OPTION:LNK
$A3 COMPILATION MODULE:$MCOB UNIT:202
$A3 COMPILATION MODULE:$APF UNIT:202
$A3 COMPILATION MODULE:MYSUBS UNIT:212
$A3 COMPILATION MODULE:<CR>
$A3 COMPILATION OPTION:<CR>
$A3 COMPILING TEST16  .....OLD VERSION DELETED
$A3 COMPILATION COMPLETED OF S.TEST16 ON UNIT 210
$A3 NO OF ERRORS  :     0
$A3 NO OF WARNINGS:     0
```

would be replaced by the following equivalent $SDL32 dialogue:

```
GSM P1 READY:$SDL32
SPEEDBASE II COMPILER VERSION 2.11
$A3 SOURCE:TEST32 UNIT:210
$A3 OBJECT  UNIT:210  SIZE:64000
$A3 DICTIONARY 1:TEST1 UNIT:210  GENERATION 37
$A3 DICTIONARY 2:<CR>
$A3 LISTING UNIT:202 SIZE:<CR>
$A3 COMPILATION OPTION:COP
$A3 COPY LIBRARY:COPY1  UNIT:211
$A3 COPY LIBRARY:COPY2  UNIT:211
$A3 COPY LIBRARY:<CR>
$A3 COMPILATION OPTION:LNK
$A3 LNK> LOAD-MODULE ID:P.MYDLMS UNIT:212
$A3 LNK> LOAD-MODULE ID:<CR>
$A3 COMPILATION OPTION:BL
$A3 COMPILATION OPTION:VER  VERSION No:V8.1
$A3 COMPILATION OPTION:<CR>
$A3 COMPILING TEST32  .....OLD VERSION DELETED
$A3 COMPILATION COMPLETED OF S.TEST32 ON UNIT 210
$A3 NO OF ERRORS  :     0
$A3 NO OF WARNINGS:     0
```

Note that:

- Dynamic Load Module (i.e. the 32-bit equivalents of 16-bit sub-routine libraries) must be specified explicitly using the "LNK" option. Note that the system DLM library (i.e. P.$SDLM0 on unit $$D) is always used by the compiler and does not have to be specified in the $SDL32 dialogue. The compiler does not use the $$DLM index files that are recognised by the loader. All application DLM libraries must be specified explicitly.

  Important note: $SDL always automatically prefixes the reply to the Compilation Module prompt with "C." (e.g. a reply of $MCOB, rather than C.$MCOB, **must** be supplied). However, $SDL32 does **NOT** automatically prefix the reply to the Load Module prompt with "P.". This slight dialogue change is deliberate to allow DLM's outside a DLM library to be specified at compile-time;

- $SDL32 includes the "VER" option. This option allows the first 6 bytes of the program title to be specified at compile time.

F.3.4.1.1 Compiler Options

The following $SDL option are not supported in $SDL32:

HIGH	(the high address is always FFFF)
BASE	(the base address is always 0000)
HT	(Help Text is always stored in the frame)

The following new option is available in $SDL32:

VER	Specify a 6-character version number

The type of modules specified when the LNK option is in effect have changed. Whereas the COMPILATION MODULE required by $SDL is a compilation module or library (e.g. C.$MCOB) the COMPILATION MODULE required by $SDL32 is a Dynamic Load Module (DLM) library (e.g. P.COMMON). Note that $SDL32 always assumes P.$SDLM0 as a Compilation Module so that there is no need to specify P.$SDLM0 in the LNK option.

F.3.4.2 Speedbase Language Differences

This section describes the language differences between 16-bit Speedbase and the 32-bit Development language.

F.3.4.2.1 Use of pointers (PIC PTR)

The PIC PTR statement now generates a 4 byte pointer. Any frames that redefine PTRs or perform PTR arithmetic will generally need to be modified. However the use of pointers in Speedbase Applications is generally very rare, and this is therefore unlikely to be an issue with the majority of programs.

F.3.4.2.2 Global Symbols

Global Symbols are made available to Dependent Frames using the 32-bit Dynamic Linking Facilities. The CONTROLLING FRAME statement continues to signify that the symbols within the compilation are all (generally) to be regarded as global. There is, however, a restriction relating to the use of Global Based Items: Where a Based item is to be declared Global, it is a requirement of $SDL32 that both the BASED item and its base PTR are locally declared items within the compilation, and that both items are Global symbols. Because a FILLER item can never be a

Global Symbol, this means that all Global Based Items must have an explicit base PTR assigned to them.

For example, when ABC is a Global Symbol, the statement:

```
77 ABC BASED FILLER  PIC ..
```

will generate a compilation error because FILLER is by definition a non-global symbol. The statement must be modified as follows:

```
77 ABCPTR   PIC PTR
77 ABC  BASED  ABCPTR PIC ..
```

where ABCPTR is a locally declared Global Symbol.

The same issue will arise if an external based global is redefined in the Data Division thus locally generating a new based global symbol. In this instance the item's base pointer, while a valid global symbol, has not been declared within the compilation, and thus will generate a compilation error.

F.3.4.2.3 Redefinition's

Earlier versions of the 16-bit $SDL contained a problem which allowed invalid redefinition's to compiled without warnings or errors. For example, the following statement:

```
77 ABC REDEFINES X OCCURS n PIC X
```

would compile without error even though the total size of the generated array exceeded the length of x.

The compiler has been corrected so that this situation now results in a warning.

F.3.4.2.4 EXEC Calls

The EXEC operation has been extended so that up to 7 parameters may now be passed with the USING clause. To provide compatibility with the Cobol implementation, an exception condition generated within the EXED'ed program is now also passed back to the calling program.

When an exception is signalled at the highest level of control in the Load, Procedure, or Unload Division, the Back-Frame ID of the optional SEQUENCE statement is examined. If the Back-Frame ID is set, then the indicated frame is loaded next, as was formerly the case, and the exception condition is cleared. Otherwise the exception condition is passed back to the EXECing program, which must therefore contain an ON EXCEPTION statement immediately following the EXEC.

Note that these exception conditions were previously NEVER passed back. Some existing applications may therefore need to be modified to trap these newly passed exceptions.

Note that the Window Division (In the absence of a Procedure Division) also generates exception conditions at the highest level of control. However, to maintain compatibility with earlier versions of Speedbase, exception generated directly by the Window Division are NOT returned via the EXEC mechanism.

F.3.4.2.5 Variable Length Character Items

When establishing a contestant text string as a character item the Picture Clause may be coded as:

PIC X(?)

The length of the data item will be set to the sum of the lengths of the VALUE clauses which follow it, and eliminates the need to count up the length of such strings. For example:

```
77   FIELD PIC X(?)
     VALUE     "INSUFFICIENT STOCK REMAINING"
     VALUE     " - CANNOT PROCESS ORDER"
```

F.3.4.2.6 New DO … ENDDO Construct

The following language construct is now allowed:

```
DO FOR X = A [TO B [STEP C]]
     * Statements to be executed while stays in range
ENDDO
```

The loop count, X, must be a computational variable. The initial value, A, must be a computational value or literal. The limit, B, must be a computational value or literal, if the TO clause is coded. The step, C, must be a numeric literal, if the STEP clause is coded.

The Variable X is first set to the Start Value in A. This is compared with the Limit in B, if specified, and provided all is well, the enclosed statement(s) are executed. When the ENDDO is encountered the value in X is increased by the Step Length, C, or 1 if no STEP clause is coded, and the processing resumes with a comparison with the Limit.

If the Step Length is not negative, or is omitted, then execution will continue provided that the value in X is not greater then the value of the Limit B. If the Step Length is negative then execution will continue provided that the value of X is not less than the value of the Limit B.

If the STEP clause is omitted a Step Length of 1 is assumed.

If the TO clause is omitted no limit checking takes place, and execution will continue until a transfer of control statement causes execution to leave the DO loop, or the variable X overflows.

F.3.4.2.7 The EDIT statement

$SDL32 recognises the EDIT statement as documented in section 4.9 of the Global Cobol Language. The EDIT statement allows a numeric value to be edited into special formats. It is coded as:

EDIT A INTO B FORMAT C

Where A is a computational or display numeric variable, or a computational literal. It can have a maximum of 12 digits preceding the decimal point, and a maximum of 6 following the decimal point. B must be a character variable with a maximum length of 30 characters. C may be a character variable or literal.

The value given in A is edited according to the format specified in C and the result is placed, right justified, in B.

The number of decimal places specified in the picture clause of A (or the number of digits following

the decimal point if a literal) determines the number of decimal places that will appear in the result.

Please refer to section 4.9 of the Global Cobol Language Manual for further details.

F.3.4.2.8 Access Methods

$SDL32 recognises all the Access Methods supported by $COBOL (including DMAM). Consequently, there is no need, as for 16-bit Speedbase applications, to write $COBOL sub-routines to include access methods (e.g. Open TFAM) that are not recognised by the Speedbase compiler.

F.3.4.2.9 Screen Formatting

$SDL32 supports the various Screen Formatting statements as documented in the Global Cobol Screen Presentation Manual.

F.3.5 Migrating $COBOL Applications

This section describes the factors that must be considered when converting 16-bit Cobol applications to 32-bit.

F.3.5.1 Migrating $COBOL Applications Overview

The 32-bit Development System is a full merger of the 16-bit Global Cobol and 16-bit Global Speedbase development systems. The 32-bit compiler, $SDL32, has incorporated all the functionality provided by $SDL and $COBOL, which $SDL32 fully supersedes. In addition to this complete functionality merger, the 32-bit Development System incorporates many new facilities such as full 32-bit Memory Management and Dynamic Link Loading, which are now available to **ALL** developers of Global software.

For existing users of 16-bit Global Cobol the 32-bit Development System offers the number of compelling benefits:

Memory Management: The 32-bit Development System eradicates the memory management and availability problems present in the 16-bit development systems. A program may now be composed of an unlimited number of executables, each of which occupies its own address space. There is now longer any limit to the amount of memory that may be used by an application program, which allows new functionality to be added to existing functions with a minimum of effort.

Dynamic Link Loading: System and Application Service routines are now dynamically loaded when required, rather than being physically linked into each application program. This dramatically simplifies software distribution and upgrades, as only explicitly amended programs need be re-installed. For example, enhancements and bug fixes to system routines no longer require changes, or re-installation, of application software thus simplifying version control.

Application Service Modules: Common application service routines can now be executed at any time, and from any applications. This important element of Object-Orientated development allows common processes to be defined, and accessed, throughout all programs of an application. The 32-bit Development System even allows an entire application to be treated as an object, to be invoked while running another completely separate application.

Industry Standard Databases: The 32-bit Development System allows existing Global Cobol applications to access databases such as Informix C-ISAM, Btrieve and SQL Server with a

minimum of development effort. Once converted, the application can access all supported database structures using a single set of source and object programs irrespective of the storage methodology employed.

GUI Capabilities: Existing Global Cobol products can now be extended to operate with the Global Windows Workstation. Each program may make use of Windows definitions, which greatly simplifies the GUI interface. It is not necessary to convert the entire application to GUI standards, the project may be addressed on s screen-by-screen basis addressing the most important elements first. The GUI facilities are compatible with older screen facilities, and thus can be implemented incrementally, allowing a gradualistic approach to product enhancement.

F.3.5.2 Cobol Application Structure

The first step in the conversion process is to identify commonly used compilands (such as common application routines) and to convert these to Dynamic Load-Modules. The order in which the Load Modules are compiled is important. The Cobol linkage editor permits two way dependencies to exist between modules (i.e. module A may reference a global symbol defined in module B, while at the same time module B may reference a global symbol in module A).

Two-way dependencies are not supported by $SDL32, which requires all references to externally defined globals to be uni-directional. It is therefore important to convert and compile the highest level and simplest compilands first. If this process reveals two way dependencies, then the simplest solution is to move the offending global symbol declarations higher into the Load-module structure.

Once all the common application functions have been converted to Load-modules, it is often convenient to combine them within a Program Library. This library is then specified in subsequent compilations of executables using the compiler LNK option.

The individual programs within the application may then be converted. A program that does not make use of overlay structures is simply re-compiled, and the external references to previously defined load-modules are automatically resolved. Some modifications may, however, be required for more complex programs that have been arranged into overlay structures.

The root module of an overlay structure will not normally require any special treatment, other than as elsewhere described in this appendix. In particular, you should **not** need to include the CONTROLLING FRAME statement in the root module. When compiling overlays, it will, however, generally be necessary to declare that the overlay is Dependent on the root frame using the DEPENDENT ON Clause of the PROGRAM statement, which is described earlier in this appendix.

The use of the DEPENDENT ON Clause causes any Global Symbols to be extracted from the root program, and thus allows them to be referenced by the overlay during compilation. In deeper structures, each overlay level will generally declare dependency on it's immediately higher level overlay. At compile time, the compiler then extracts the global symbols from **all** higher overlays within the structure, including the ultimate root program.

In order to squeeze the last possible byte out of available memory, a number of Cobol programs have been forced into highly unusual and non-standard overlay structures. This has sometimes been done using dubious techniques such as hard-linking modules at particular addresses, or assuming the continued presence of loaded overlays following program termination. Any dubious overlay structures should therefore be rationalised as part of the conversion task.

It should be noted that, unlike the 16-bit Cobol implementation, an overlay is automatically unloaded from memory once execution of that overlay completes. If the overlay needs to be re-executed, it

must first be reloaded using the EXEC or Load verbs. Note that the overlay entry-point ($$EPT) will be changed following each reload.

F.3.5.3 Compiler Dialogue Changes

Whereas the dialogue for $SDL32 is deliberately very similar to the dialogue for $SDL (see section F.3.4.1, above) the dialogue for $SDL32 is very different from the dialogue for $COBOL/$LINK.

The most striking difference between the $COBOL and $LINK combination and $SDL32 is that $SDL32 is a combined compiler and a linker (although the actual linking takes place at program load/resolve time – the compiler must still be aware of the external dependencies). This difference in functionality leads to a slightly different dialogue. The following example dialogue for $COBOL and $LINK:

```
GSM P1 READY:$COBOL
$43 SOURCE:TEST16  UNIT:210
$43 COMPILATION UNIT:210 SIZE:64K
$43 COPY LIBRARY:S.COPY1  UNIT:211
$43 COPY LIBRARY:S.COPY2  UNIT:211
$43 COPY LIBRARY:<CR>
$43 LISTING UNIT:202
$43 COMPILER OPTION:BL
$43 COMPILER OPTION:<CR>
$43 COMPILING
$43 END OF FIRST PASS
$43 NUMBER OF ERRORS      0
$43 NUMBER OF WARNINGS    0
$43 COMPILATION OF S.TEST16 UNIT 210 COMPLETED
GSM P1 READY:$LINK
$44 LINK:TEST16      UNIT:210
$44 LINK:C.MYSUBS    UNIT:212
$44 LINK: <CR>
$44 PROGRAM:TEST16   UNIT:210
$44 LISTING UNIT:202
$44 LINK OPTION:<CR>
$44 LINKAGE EDITING
$44 LINKAGE EDIT COMPLETE
```

would be replaced by the following equivalent $SDL32 dialogue:

```
GSM P1 READY:$SDL32
SPEEDBASE II COMPILER VERSION 2.11
$A3 SOURCE:TEST32 UNIT:210
$A3 OBJECT  UNIT:210  SIZE:64000
$A3 DICTIONARY 1:<CR>
$A3 LISTING UNIT:202 SIZE:<CR>
$A3 COMPILATION OPTION:COP
$A3 COPY LIBRARY:S.COPY1  UNIT:211
$A3 COPY LIBRARY:S.COPY2  UNIT:211
$A3 COPY LIBRARY:<CR>
$A3 COMPILATION OPTION:LNK
$A3 LNK> LOAD-MODULE ID:P.MYDLMS UNIT:212
$A3 LNK> LOAD-MODULE ID:<CR>
$A3 COMPILATION OPTION:BL
$A3 COMPILATION OPTION:<CR>
$A3 COMPILING TEST32  .....OLD VERSION DELETED
$A3 COMPILATION COMPLETED OF S.TEST32 ON UNIT 210
$A3 NO OF ERRORS  :     0
$A3 NO OF WARNINGS:     0
```

Note that:

- The reply of <CR> to the DICTIONARY: prompt by-passes the Speedbase Database Dictionary dialogue;

- Copy libraries must be specified explicitly using the “COP” option. Note also that, whereas $COBOL uses a default unit of $CL for the copy libraries, $SDL32 offers no default unit;

- Dynamic Load Module (i.e. the 32-bit equivalents of 16-bit sub-routine libraries) must be specified explicitly using the “LNK” option. Note that the system DLM library (i.e. P.$SDLM0 on unit $$D) is always used by the compiler and does not have to be specified in the $SDL32 dialogue. The compiler does not use the $$DLM index files that are recognised by the loader. All application DLM libraries must be specified explicitly;

- Whereas $COBOL allows the size of the Listing File and Object File to be specified in Kb (e.g. 32K), $SDL32 only accepts numeric responses (e.g. 32768);

- $SDL32 includes the “VER” option. This option allows the first 6 bytes of the program title to be specified at compile time.

F.3.5.4 Language Considerations

F.3.5.4.1 ENDSOURCE Statement

To provide compatibility with 16-bit Speedbase frames the ENDSOURCE statement must be added to the end of every 32-bit source. Thus, the general structure of a 32-bit source is:

```
PROGRAM
DATA DIVISION
[LINKAGE SECTION]
[EXTERNAL SECTION]
[COMMON SECTION]
PROCEDURE DIVISION
ENDPROG
ENDSOURCE
```

Please refer also to sections F.3.2.5 and F.3.2.6 for details of the additional LOCAL and GLOBAL sections.

F.3.5.4.2 DISPLAY and ACCEPT Verbs

$SDL32 normally expects to process the standard Speedbase Accept and Display (and related) verbs. In order to simplify migration of 16-bit Cobol programs containing the $COBOL variants of these verbs, a new compiler option, CB, has been incorporated. This option allows the compiler to process all ACCEPT and DISPLAY statements using the $COBOL format. This permits statements such as:

```
ACCEPT A LINE l COL c NULL ...
```

to be compiled without modification.

F.3.5.4.3 DISPLAY SPACE and DISPLAY SPACES

$SDL32 doesn't recognise either of the following statements:

```
DISPLAY SPACE
DISPLAY SPACES
```

The following statement is equivalent and must be used instead of either of the above:

```
DISPLAY ""
```

F.3.5.4.4 Use of Pointers (PIC PTR)

The Data Division PIC PTR statement now generates a 4byte pointer to facilitate 32-bit addressing. Some care will be needed when recompiling existing 16-bit programs to ensure that any redefinition's do not assume the 2 byte format. The highly dubious (and strongly discouraged) practice of performing arithmetic on PTRs will generally work if the last two bytes of PTR are redefined as a 9(4) Comp item, and all store operations are performed using the intermediate code Store Unsigned ($STUS) instruction. Note that a 32-bit subroutine, CPTR$, is available to perform arithmetic on 32-bit pointers.

A related coding technique that may require some attention is the use of Uninitialised Storage, which is also not supported by $SDL32. The simplest solution is again to move the data area into a higher level overlay, and then declare the data items within it as global.

F.3.5.4.5 $COBOL Language verbs

The following $COBOL language verbs are NOT supported by $SDL32:

```
COMPUTE
CHAIN
RUN
FUNCTION
```

All other $COBOL language verbs, as documented in the Global Cobol Language Manual, are supported by $SDL32.

At the time of writing, the PRIVILEGED option is not supported. Consequently, no 32-bit application can prevent the ^W Break facility from being used.

None of the $OPT compile time options are supported by $SDL32.

F.3.5.4.6 Length of the FD has increased

The length of the FD control block generated by the FD verb has been increased by 12 bytes (to allow for 3 32-bit PIC PTR fields). Any redefinition's of FD's should take this extra length into account (see sections F.3.5.4.7, F.3.5.4.8 and F.3.5.4.9, below).

In addition, a new flag field in the FD, FDFLAG, is set to 0 for a 16-bit FD; and –1 for a 32-bit FD. In the extremely unlikely event of a program defining an FD that is not automatically generated by the $SDL32 compiler, care must be taken to set FDFLAG to –1. Note that when the 32-bit compiler generates an FD as a result of a language statement such as:

```
FD    MYFD        ORGANISATION R-S
```

```
etc.
```

the FDFLAG field is always initialised correctly to –1.

F.3.5.4.7 TFAM FD Redefinition's

As explained in section F.3.5.4.6, the length of the fixed portion of the 32-bit FD is 12 bytes longer than the equivalent 16-bit FD. This extra length alters the redefinition's required for the TFAM Access Method.

For 16-bit TFAM the recommended coding technique is:

```
FD    FDOTFA     ORGANISATION OR$83
etc.
01    FILLER REDEFINES FDOTFA
 02   FILLER PIC X(86)                  * Allow for 16-bit FD
 02   TXLLN PIC 9(4) COMP
 etc.
```

For 32-bit TFAM this must be replaced by:

```
FD    FDOTFA     ORGANISATION OR$83
etc.
01    FILLER REDEFINES FDOTFA
 02   FILLER PIC X(98)                  * Allow for 32-bit FD
 02   TXLLN PIC 9(4) COMP
 etc.
```

F.3.5.4.8 Open TFAM FD Redefinition's

As explained in section F.3.5.4.6, the length of the fixed portion of the 32-bit FD is 12 bytes longer than the equivalent 16-bit FD. This extra length alters the redefinition's required for the Open TFAM Access Method.

For 16-bit Open TFAM the recommended coding technique is:

```
ORGANISATION OR$83O TYPE 3 EXTENSION 544
FD    FDOTFA     ORGANISATION OR$83O
etc.
01    FILLER REDEFINES FDOTFA
 02   FILLER PIC X(86)                  * Allow for 16-bit FD
 02   TXLLN PIC 9(4) COMP
 etc.
```

For 32-bit Open TFAM this must be replaced by:

```
ORGANISATION OR$83O TYPE 3 EXTENSION 552
FD    FDOTFA     ORGANISATION OR$83O
etc.
01    FILLER REDEFINES FDOTFA
 02   FILLER PIC X(98)                  * Allow for 32-bit FD
 02   TXLLN PIC 9(4) COMP
 etc.
```

Furthermore, if the advanced technique to call via the Open TFAM Access Method Pointer directly is being used to access Open TFAM files with a line length greater than 256 bytes, the FD redefinition to expose the FDPTR pointer must be changed form:

```
01    FILLER          REDEFINES FDOTFA
 02   FDPTR     PIC PTR                 * 16-bit  pointer
```

to:

```
01    FILLER          REDEFINES FDOTFA
 02   FD16PTR   PIC SPT                 * 16-bit pointer (ignored)
 02   FILLER          PIC X(78)               * Reset of the FD
 02   FDPTR     PIC PTR                 * 32-bit  pointer
```

F.3.5.4.9 Open Direct FD Redefinition's

As explained in section F.3.5.4.6, the length of the fixed portion of the 32-bit FD is 12 bytes longer than the equivalent 16-bit FD. This extra length alters the redefinition's required for the Open Direct Access Method.

For the 16-bit Open Direct Access Method the recommended coding technique is:

```
FD    FDODIR    ORGANISATION OR$98
etc.
01    FILLER          REDEFINES FDODIR
 02   FILLER          PIC X(88)               * Allow for 16-bit FD
 02   PANAME    PIC X(99)               * Pathname
 etc.
```

For the 32-bit Open Direct Access Method this must be replaced by:

```
FD    FDODIR    ORGANISATION OR$98
etc.
01    FILLER          REDEFINES FDODIR
 02   FILLER          PIC X(100)              * Allow for 32-bit FD
 02   PANAME    PIC X(99)               * Pathname
 etc.
```

[ALSO MENTION THAT OR$98W -→ OR$98]

F.3.5.4.10 32-bit Screen Formatting

All 16-bit Screen Formatting maps (e.g. C.MYMAP) must be converted to 32-bit MAP DLM's. This is simply achieved by running all 16-bit maps through the $FORM32 utility. In addition to converting 16-bit maps to 32-bit Map DLM's, $FORM32 can also be used to create and maintain 32-bit Map DLM's. The dialogue for $FORM32 is deliberately based on the dialogue for $FORM.

Note that a single 32-bit map DLM may contain several Screen Formatting maps.

F.3.5.4.11 Format of the MD control block has changed

The format of the 32-bit MD control block generated by the MD verb has changes completed between 16-bit and 32-bit applications. No application should make any assumptions regarding the internal format of the MD block.

F.3.5.4.12 32-bit Interface to SVC-61

The 32-bit calling interface to SVC-61 is different from the 16-bit calling interface (as documented in the V8.1 File Converters manual).

F.3.5.4.12.1 32-bit Interface to SVC-61 for GSM (Windows)

The format of the GSM (Windows) SVC-61 DS-block for 32-bit applications is as follows:

```
01    DS
 02   DSFUNC      PIC X            * Function code (top bit set)
 02   DSMODE      PIC X            * Subfunction or mode
 02   DSRES            PIC 9(4) COMP    * Windows NT return code
 02   DSHAND      PIC 9(4) COMP    * File handle (not used)
 02   DSNAME      PIC SPT          * Reserved for future use
 02   DSBUFF      PIC SPT          * Reserved for future use
 02   DSATTR      PIC X(2          * File attributes
 02   DSNBYT      PIC 9(4) COMP    * Number of bytes moved
 02   DSPAR1      PIC X(2)         * Function specific
 02   DSPAR2      PIC X(2)         * Function specific
 02   DSPAR3      PIC X(2)         * Function specific
 02   DSPAR4      PIC X(2)         * Function specific
 02   DSHA32      PIC 9(9) COMP    * Win-32 file handle
 02   DSHAFI      PIC 9(9) COMP    * Win-32 find handle
 02   DSRES32     PIC 9(9) COMP    * Win-32 error code
 02   DS32NAME    PIC PTR          * 32-bit ptr to file name
 02   DS32BUFF    PIC PTR          * 32-bit ptr to buffer
 02   FILLER           PIC X(3)         * Reserved for future use
 02   DS32ERR     PIC X            * 32-bit error code
```

The top-bit of the function code, in DSFUNC, must be set to indicate a 32-bit operation (and thus a 32-bit format DS-block). For example, the function code for the 16-bit “Open File” operation code is #3D. The function code for the equivalent 32-bit operation is #BD (i.e. #3D + #80).

The 32-bit pointer, DS32NAME, replaces the 16-bit DSNAME.

The 32-bit pointer, DS32BUFF, replaces the 16-bit DSBUFF.

The special result code of 100 (in DSRES) indicates a 32-bit address error (i.e. the address in either DS32NAME or DS32BUFF is invalid). The extended error code is returned in DS32ERR:

‘N’ Page not allocated
‘S’ Block length too long for fixed length block
‘T’ String length too long for zero-terminated string
‘U’ Reserved for User Number inconsistency
‘V’ Page type is inappropriate

‘W’ Page number out of bounds (too high)
‘X’ Page table internal inconsistency
‘Y’ Page number out of bounds (too low)

F.3.5.4.12.2 32-bit Interface to SVC-61 for GSM (Unix)

The format of the GSM (Unix) SVC-61 DS-block for 32-bit applications is as follows:

```
02    DS
 02   DSFUNC              PIC 9(4) COMP      * Function code (+ 4000)
 02   DSRECN              PIC S9(9) COMP     * C-ISAM isrecnum
 02   DSSTA1              PIC 9(2) COMP      * C-ISAM isstat1
 02   DSSTA2              PIC 9(2) COMP      * C-ISAM isstat2
 02   DSKPART             PIC 9(4) COMP      * C-ISAM max number of key parts
 02   DSLNID              PIC X              * System ID
 02   DSUSER              PIC 9(2) COMP      * User number
 02   DSEXTR              PIC X(10)          * Specialised data
 02   DSERR               PIC S9(9) COMP     * Result code errno
 02   DSDATA              PIC SPT            * Unused
 02   DSSIZE              PIC 9(4) COMP      * Max size of return data
 02   DSRET                    PIC X(4)           * Returned value
 02   DSPAR OCCURS 6      PIC X(4)           * Up to 6 parameters
 02   DS32DAT             PIC PTR            * Pointer to return data
 02   DS32ERR             PIC X              * 32-bit error code
```

A value of 4000 must be added to the equivalent 16-bit function code, in DSFUNC, to indicate a 32-bit operation (and thus a 32-bit format DS-block). For example, the function code for the 16-bit “opendir” operation code is 10. The function code for the equivalent 32-bit operation is 4010.

The 32-bit pointer, DS32DAT, replaces the 16-bit DSDATA.

The special result code of 100 (in DSERR) indicates a 32-bit address error (i.e. the address in DS32DAT is invalid). The extended error code is returned in DS32ERR:

‘N’ Page not allocated
‘S’ Block length too long for fixed length block
‘T’ String length too long for zero-terminated string
‘U’ Reserved for User Number inconsistency
‘V’ Page type is inappropriate
‘W’ Page number out of bounds (too high)
‘X’ Page table internal inconsistency
‘Y’ Page number out of bounds (too low)

F.3.5.4.13 Interface to SPD

The LOAD$, RESID$ and URESI$ routines have not be converted to 32-bit. Consequently, any 16-bit program that load an “assist module” on the User Stack or System Stack must be modified. The dynamic and inherently relocatable nature of the 32-bit run-time environment removes the requirement for the special, relocatable 16-bit User Stack/System Stack. However, whereas relocatable 16-bit Cobol modules can be trivially converted to (inherently relocatable) 32-bit DLM’s special action must be take for the Serial Port Driver (SPD) module.

The SPD modules for GSM (BOS) (e.g. %.J5S03, for example) and GSM (DOS) (e.g. %.JWS03, for example) are both relocatable 8086 assembler modules that MUST be loaded on the User Stack or System Stack in order to allow the Commercial Code Interpreter to execute their 16-bit assembler entry-points.

The SPD module for GSM (Windows) (i.e. %.W1S) is actually a “dummy” data-only module that merely cause the Commercial Code Interpreter to execute the true entry point in the GLOBAL.EXE module (i.e. the SPD handler for Windows NT is always permanently available). Similarly, the SPD module for GSM (Unix) (i.e. %.C2S) is actually a “dummy” data-only module that merely cause the Commercial Code Interpreter to execute the true entry point in the glintd module (i.e. the SPD handler for Unix is always permanently available). Because the actual SPD entry point is always available in the GSM (Windows) and GSM (Unix) nucleus, a new SVC entry point (SVC 84) has been allocated for SPD. The calling convention for 32-bit SVC 84 is identical to the calling convention of the 16-bit SPD entry point address (established in $$EPT by the 16-bit LOAD$ routine). The SPD interface for GSM (Windows) and GSM (Unix) is beyond the scope of this document.

F.3.6 New 32-BIT STOP and EXIT Codes

F.3.6.1 New STOP codes returned by the 32-bit Loader

The following new STOP codes may be returned by the 32-bit Loader:

STOP 109 An attempt has been made to load a 32-bit program but there was insufficient memory available

STOP 110 An unexpected error has occurred when allocating memory space to load a 32-bit program. This may be due to program file corruption

STOP 111 An error has occurred in the loader when it attempted to relocate symbols. Either the referenced program is not loaded or the referenced symbol is not present

STOP 112 An attempt has been made to run a 32-bit Dynamic Load Module directly

STOP 113 A 32-bit Dynamic Load Module expected by a 32-bit program is not present on the Dynamic Load Module unit

STOP 114 An attempt has been made to load a 32-bit program that has had a compilation error

STOP 115 An attempt has been made to load a 32-bit Dynamic Load Module which has suffered compilation error

STOP 116 The loader has been unable to resolve the load link chain for a 32-bit module. This may be due to program corruption

STOP 117 There is insufficient memory to allocate to load application DLM’s

STOP 118 An attempt has been made to load a 32-bit Dynamic Load Module (DLM) which has not been compiled as a DLM

STOP 119 Internal error loading from an Extended 1000-entry library (unable to load library index page)

F.3.6.2 New EXIT codes returned by the 32-bit Loader

The following new EXIT codes may be returned by the 32-bit Loader:

EXIT 109 An error has been detected when attempting to execute a 32-bit program;

F.3.6.3 New STOP codes returned by the 32-bit Executive Interface

The following new STOP codes may be returned by the 32-bit Executive Interface:

STOP 1001 No External Pointer Table available

STOP 1002 External Pointer entry not allocated

STOP 1003 External Pointer page number invalid

STOP 1004 External Pointer page table entry invalid

All of these STOP codes indicate an internal error within Global System Manager. They are *unlikely* to be caused by a bug in application software.

F.3.6.4 Resolver error codes returned by the 32-bit Resolver/Loader

The STOP 116 message will be preceded by an error code and diagnostic message returned by the Resolver/Loader. For some of these error conditions the name of the failing symbol will also be displayed:

Code	*Message*
@	Unable to find SVC page
A	Symbol table link mismatch
B	Symbol table size mismatch
C	Symbol table not resolved
D	Symbol table empty
E	Linked symbol table mismatch
F	Linked symbol table size error
G	External symbol table mismatch
H	Symbol number too high
I	Symbol number out of range
J	External symbol not found
K	Symbol mismatch (type)
L	Base pointer symbol too high
M	Base pointer symbol too high
N	Base pointer symbol mismatch
P	Symbol page not allocated
Q	Symbol number too high
R	No code/data page allocated
S	Infinite recursion detected
T	Indirect page number too high
U	Error resolving IN-block
V	Symbol mismatch (format)
W	Symbol mismatch (length)

X Symbol mismatch (occurs)
Y Symbol mismatch (group length)
Z DBID mis-match *symbol1* and *symbol2*
a Indirect page number invalid
b Indirect page not allocated
c Indirect page too small
d Indirect pointer invalid
e Indirect symbol number too high
f Invalid resolver recursion
g IN-block symbol number too high
h 1st DBID symbol too high
i 1st DBID symbol wrong type
j 2nd DBID symbol too high
k 2nd DBID symbol wrong type
w Compiled with pre-V2.03 $SDL32
x Indirect page number invalid
y Compiled with pre-V2.00 $SDL32
z Cannot find assoc. code page

F.3.6.5 New STOP codes returned by the 32-bit Speedbase Presentation Manager

STOP 25001 An attempt has been made to execute (EXEC) a Load module.

STOP 25002 The Overlay level is deeper than 16 levels.

STOP 25003 An Invalid Frame ID has been passed to B$CHAN

STOP 25004 Attempted recursive use of Intercept routine $PIFIN

STOP 25006 Internal system error detected by the Resolve/Relocate SVC. Speedbase PM displays the operation code and result code of the failing SVC-79 operation before terminating the application

STOP 25007 Internal system error detected by SWAP Page handler

STOP 25008 Internal system error detected by Qualifier translator B$QLN

STOP 25009 More than 64 link stack items saved; This error will occur if a succession of Load Division EXEC USING statements cause more than 64 link stack items to be saved. These parameters are always passed to the Procedure Division of the EXED'ed program, and thus must be saved if the EXED'ed program contains a Load Division. If the Load Division itself executed a further EXEC USING, then these parameters must also be saved (and so forth). It is therefore theoretically possible (but highly unlikely) that this limit is exceeded.

STOP 25010 Invalid opcode passed to BA$MEM

STOP 25011 Generation Number mis-match. This STOP CODE will occur if an attempt is made to load two, or more, programs that were compiled using different Generation Numbers of the same database. The Speedbase Frame Loader displays the Database ID and Program Names of the mis-matching programs before terminating the application.

F.3.7 The 32-bit "Same Partition" Debugger

This section describes an initial, simple debugger that allows 32-bit programs to be debugged. A vastly more sophisticated and functional 32-bit debugger is currently being designed.

Important Note: The current 32-bit debugger, $DBG32, is intended as a development tool – it does not currently include a “user friendly” Diagnostics Screen. When a 32-bit exception (e.g. EXIT WITH 1) occurs, control passes directly to the $DBG32 command prompt. At the time of writing, an advanced windows debugger is being implemented.

The “same partition” debugger operates in simple scroll mode and supports the following commands (described in alphabetic order):

Address parameters may be given in either numeric or symbol formats. The numeric format is an 8-digit hexadecimal address consisting a 2-byte page number and a 2-byte offset, *ppppoooo*. For example:

006F5634

refers to offset 5634 (hex) in page 006F (hex).

The symbolic format is *module_name/symbol_name*. For example:

MYPROG/Z-WORK

refers to the symbol Z-WORK in frame or program MYPROG.

F.3.7.1 C - cancel trap

This instruction cancels the trap on the address as given as a parameter. If no parameter is passed then it cancels the current trap. If a trap-address parameter is given the trap is unset (see details of the Trap instruction below).

A special reply of C<CTRL B> will clear all the traps set by the current user.

F.3.7.2 D - Display DLA information

This instruction displays the 32-bit Diagnostic Logout Information. The information returned includes the STOP or EXIT code, the current Accumulator value, the Link Stack (showing the Call and Return addresses) and details of the last program loaded and the last file access.

F.3.7.3 H -Display Help information

This instruction merely displays a Help screen listing all the commands available in $DBG32.

F.3.7.4 I/M Inspect/Modify data

These instructions take an address and format as parameters as specified below.

The allowed formats available are:

Cn, Cn,m, C, Xn, x, H, Hn, P, D

If no format is supplied then a format of H16 will be assumed. Up to 16 bytes can be limited in hex mode.

F.3.7.5 L - Load program

This instruction loads, and fully resolves the addresses in a 32-bit program. Once the program has been loaded into memory, subsequent operations (e.g. T, R etc.) can be used during the debug session.

F.3.7.6 N - Symbol Table information

This instruction displays the symbol table information for a symbol given in *module_name/symbol_name* format.

The 32-bit address of the symbol can be used for subsequent instructions (e.g. T hhhhhhhh).

F.3.7.7 O- Overlay trap

This instruction sets the overlay trap field in the system area to the overlay specified. The trap all overlays option is not allowed.

F.3.7.8 P – Print Diagnostic Information

This instruction prints a debug report using the output of the D, W and Y commands.

F.3.7.9 Q - Quit the debugger

The quit instruction merely exits the debugger and terminates and unloads the 32-bit program.

F.3.7.10 R – Resume

This command resumes 32-bit program execution.

F.3.7.11 S – Scroll

This instruction merely toggles between scrolled and formatted screen mode.

F.3.7.12 T - trap

This instruction sets a trap at the specified address.

F.3.7.13 V - Value trap

This instruction has not been fully implemented.

F.3.7.14 W – List Page Table

This instruction lists the memory pages currently allocated for the program being debugged.

F.3.7.15 X – List Trap Table

This instruction list the trap table for the program being debugged. Trap table entries will be removed when the current program terminates.

F.3.7.16 Y - Dump virtual machine

This instruction dumps the registers from the DLA block. This instruction is intended for internal use only.

F.3.7.17 Z - Get 32-bit address of program

This instruction returns the code page number of the specified program.

F.4 An Important Note on Large 32-bit Programs

In the V8.1j $SDL32 all locally declared global variables were treated as referenced, and hence ate away at the 2040 referenced symbol limit for each program. The V8.1k $SDL32 only counts actual Procedure and Window Division references towards the referenced symbol limit, which means that thousands of global symbols can be declared without hitting any compiler restrictions. The compiler sort routine has also been rewritten; the V8.1j, rather primitive bubble sort, caused unacceptable overhead when dealing with larger programs. In the process, the general performance of the V8.1k compiler has improved by up to an order of magnitude.

The V8.1k $SDL32 is capable of dealing with very large numbers of Global Declarations. However, in order to produce an application that runs efficiently, it is in your interests to keep the number of Global Symbol Declarations to a minimum. Any Load-Module that contains more than
a couple of dozen declarations should be carefully examined. Certainly any program that declares 100's (or worse still) 1000's of Global Symbols is extremely suspect.

Even though it does not impact on any compiler limits, programs declaring large numbers of Globals use far more memory, and take considerably longer to load than necessary. For DLMs and Executables the usual problem is that the ACCESS statement does not contain a LOCAL clause, and hence all fields within the record type immediately become Global.

The CONTROLLING FRAME clause should also be looked at carefully. This clause has been carried over from 16-bit Speedbase for compatibility reasons, and causes ALL symbols in the compilation to be Global. Where possible, the CONTROLLING FRAME statement should be removed, and the data items that need to be global moved into a GLOBAL SECTION. Any Global
Entrypoints can then be recoded as ENTRY statements, and any I/O channels that need to be Global recoded as such, with the remainder coded using the ACCESS LOCAL variant.