PROGRAMMER'S CODING GUIDE

July 22, 1997

TABLE OF CONTENTS

Introduction:

This document provides a guideline for all DataBeam developers to follow. It is intended to serve as a company coding standard, which is to be strictly adhered to when generating code. It is not intended to stifle creativity, but rather to provide a standard format for all programmers to use and recognize. The main goal of this document is to provide a mechanism whereby DataBeam developers can write code which is consistent across toolkits. This provides source code customers as well as ourselves the ease of readability. Some of the guidelines discussed have various forms. The key concept to keep in mind is consistency.

It is understood that this standard does not cover every possible situation. In the event that another area is identified for addition to the standard, an extension to the standard could be submitted to or requested from SESB.

Purpose of the Standard:

The primary goals of this coding standard, in order of importance, are to enhance:

- consistency in code across toolkits
- human readability of the source code
- program maintainability
- program portability

Consistency in code:

The standard will enhance consistency by:

- providing a solid guideline for coding standards
- stressing the importance of consistent coding styles

Enhancing human readability:

The standard will enhance human readability by:

- providing a consistent format for coding each language element
- providing some white space within the source code
- minimizing the use of side effects

Enhancing program maintainability:

Program maintainability will be enhanced by:

- providing a consistent format--maintainers won't need to adapt to a large number of dissimilar styles
- utilizing features of the ANSI C/C++ standard such as function prototypes and type qualifiers to allow static type checking
- the substitution or elimination of error-prone language constructs

Enhancing portability:

Program portability will be enhanced by:

- not depending on internal operating system or compiler data formats
- specifying conformance to the ANSI standard
- promoting independence from environment-specific data sizes

Written for the reader:

As previously stated, the four primary goals of this standard are to enhance consistency, human readability, maintainability, and portability. This often means more typing (longer names, more comments, etc.) for the writer of the code. However, any significant piece of code is read by many more people than the authors (for example, it must be read and understood by the reviewers and maintainers). WRITTEN ONCE - READ A THOUSAND TIMES

Compliance with the standard:

While there may never be total agreement as to what is "the best" standard, there should certainly be agreement that following "an acceptable" standard has significant benefits.

Know the ANSI standard:

Become familiar with the standard language and library and use whenever possible. Do not re-implement standard functions, macros, or constants using the same names.

Overview:

C/C++ programs should be cleanly divided into a module per class or related functions. The module should contain all related types and data. A module should consist of two parts:

- the interface - an external specification that describes the class defined in the module
- the implementation - the actual functions and data that match the external specification

In C++, the interface corresponds to a ".h" file and the implementation corresponds to a ".cpp" file.

In C, the interface corresponds to a ".h" file and the implementation corresponds to a ".c" file.

The interface file:

The interface file (a special case of a header file) has the following contents:

- Other interface files needed to compile the current interface file
- declarations of any types and/or macros needed to use the defined class/functions

The implementation file:

The implementation file has the following contents:

- inclusion of required interface files
- declaration of local types, and/or macros
- definition of class functions

In cases where the implementation file is very large (requires judgement call) then the implementation file can be split However the interface .h file should still contain only external specifications and not be polluted with local definitions.

Multiple implementation files for a single interface should be the exception, not the rule. A very large implementation file could be a module needing a redesign.

Any other module that uses the facilities made available by a class should include the associated interface file.

Note: An implementation file should never contain "extern" declarations or re-declare a type from another implementation file. These should be imported by using a "#include" for the appropriate interface file(s).

Data Type Extensions:

DBBoolean data type:

All variables that hold boolean values, and all functions that return boolean values, can be declared as type "DBBoolean".

```
Example:

DBBoolean Timeout_Occurred = FALSE;
```

Boolean constants:

The following constants are intended to be used with the "DBBoolean" data type (described above). Each constant has either a "true" (non-zero) value, or "false" (zero) value, as shown in the following table.

"TRUE" VALUES	"FALSE" VALUES
TRUE	FALSE
YES	NO
ON	OFF

End-of-string constant:

The constant "EOS" can be used whenever you want to refer to the '\0' character at the end of a string.

Warning: "EOS" is not the same as "NULL", and their use is not interchangeable. "EOS" is the end-of-string character while "NULL" is the null pointer.

"Portable" DataBeam typedefs:

Use the typedefs defined in databeam.h (see Appendix A) for variable definitions. The purpose of these typedefs is to facilitate porting efforts in the future.

Example:

```
UShort    error_count;
```

Global data definitions:

In general, the use of global variables is discouraged. Using function parameters properly usually makes global data unnecessary. Where global data is unavoidable, use the following order for definitions in the implementation file:

- program-global variables (scope goes beyond the current module)
- file-global variables (scope is within the module) i.e. static variables

Function prototypes:

The function prototypes section contains function prototypes for all global and local functions in the file. It serves as a table of contents for the module and satisfies the forward declaration requirements of functions, where necessary. Specifying all prototypes in this section gives the reader a clear idea of the contents and organization of the module. The prototypes should be listed in the same order in which the functions are defined in the file.

Guidelines for function size:

Functions should be kept relatively short. This makes it easier for reviewers and/or maintainers to read and understand the functions. Indications that a function might be too long include:

- a length of greater than 100 lines (approximately two pages)
- heavy use of localized variables whose scope is less than the entire function
- conditional and/or loop statements nested more than four levels.

Even when the processing is linear, it often helps if the function is broken into separate pieces.

Example:

```
Int main (
     Int        argument_count,
     PChar      arguments[])
{
     InitializeData (argument_count, arguments);
     ProcessFiles ();
     CleanUp ();

     exit (EXIT_SUCCESS);
}
```

Scope of function definitions:

Declare functions using the smallest possible scope. If a function is used in only one module, declare it as "static". If the function is referenced outside of the module in which it is defined, declare it as "extern".

Function parameters:

Function parameter lists should use the ANSI format which includes both the parameter name and parameter type. Empty parameter lists should be declared as "(Void)".

A function with more than one parameter is to have one parameter per line with the types and names aligned in columns.

Examples:

```
UShort     MyBigProcedure (
                const Long      my_long,
                const MyType    &my_ptr);
```

```
Void        MyRoutine (
                  const MyType    &my_structure,
                  const Long      my_long_integer);

MyVeryLongTypeName MyVeryLongFunctionName (
                  const ParameterType1 input_data,
                  const Short          in_out_data,
                  PUChar               out_data);

Void        MyFunction (
                  const Char      my_char,
                  Char            *return_char,
                  DBBoolean       is_valid = TRUE);
```

For consistency, function parameters should be listed in the order of input parameters followed by output parameters. The final parameters should be those with default values.

Note: Avoid functions with more than five parameters or op-code arguments, where one argument determines the number, type, and function of others.

Implicit parameters:

An implicit parameter is any data object referenced in the function which is not in the parameter list and not local to the function. These should be documented by giving the type, name, origin, and description of each implicit parameter to the function. This section is organized similar to the "formal parameters" section, except for the addition of the type and origin of the data object.

Comments:

Comments should be used to describe what is going on in the source code when it is not obvious by looking at the code directly. Note that well-named types, constants, variables, and functions will often negate the need for extensive comments.

There are two general types of comments: block comments and line comments.

Block comments:

Block comments are formatted as follows:

```
/*
 * First line of the block comment
 *   Second line of the block comment
 */

/*
**   First line of the block comment
**   Second line of the block comment
*/

or

//
//   First line of the block comment
//   Second line of the block comment
//
```

Line comments:

Line comments are to be used on the trailing edge of a line of code or a declaration such as:

```
Short my_short; /*line comment for declarations */

my_short = x / y * z;  /*line comment in code */

alpha = beta | 1; // line comment in source
```

Naming Conventions Overview:

Descriptive names are an important aid to reading and understanding code. Names can describe the semantics of data or functions and may contain tips as to the type of data and where data or functions are defined. The rules defined in this document are intended to make code more readable and easier to maintain.

Variables should have the same use and meaning throughout a program. Never redefine names in inner blocks or re-declare global names within a function.

Abbreviating names:

The C language allows identifiers to be 31 characters long so there is no reason for having short, cryptic names. Names should not be recklessly abbreviated to save typing. If you abbreviate to save one or two characters, don't! Appendix D contains a list of accepted abbreviations. For consistency only these abbreviations should be used.

Functions, function-like macros, typedef's and class names:

Format: The format of a function, function-like macro, structures, unions, enumerations and type names is mixed case with no separation of words within the name. Each word in the name is capitalized. (A function-like macro is a macro that has arguments.)

Examples:

```
typedef enum Boolean {TRUE = 1, FALSE = 0};

typedef enum TropicalFruits
{
     PINEAPPLE,
     ORANGE,
     LIME
};

extern Void OutputCharacter (Char out_char);

class MyClass
{
     ...
};
```

Function and function-like macro names should generally sound like actions or operations and should be of the form <Verb><Object>.

```
Examples:

PrintErrorMessage ("Error Message");
CloseWindow (current_window);
```

Where sensible a prefix can be put on a function name that designates the module that contains that function. An example would be ComRoutineName which is in the communication bindings module.

Functions that return a transformed value of an argument should describe the transformation.

```
Examples:

GetStringLength ("String");
ConvertBufferToString (buffer);
```

Functions that return a boolean value should begin with or contain a to-be verb.

```
Examples:

IsBufferEmpty (buffer);
TokenIsValid (token);
```

Global Variables and Statics:

A global variable name is mixed case with underscores separating the words within the name. Each word in the name is capitalized.

```
Examples:

static Short Node_Count;

extern PChar Output_Character;
```

Function-local (auto) variables, structure members:

Format: The format of a function-local variable name is all lower case with underscores separating the words within the name.

```
Examples:
```

```
temp_length
saved_count
```

Usage for variables: Variable names should usually sound like proper names and be of the form <Adjective><Object>.

```
Examples:

node_count
string_length
current_window
saved_state
```

Class member variables:

A global variable name is mixed case with underscores separating the words within the name. Each word in the name is capitalized.

```
Example:

UInt                    Number_Of_Retries;
Char                    *Dial_String;
```

Constants:

Constants are names whose value is known at compile-time and whose value never changes. Constants are defined using either the "#define" preprocessor directive or as an enumeration value.

Format: Constant names appear in all upper case with underscores separating individual words in the name.

```
Example:

#define NUMBER_WINDOWS    20
#define MAXIMUM_RETRIES   10
Const Char THE_STRING[ ]="This is a constant
string";
```

Variable Declarations:

General guidelines:

Variable declarations should be limited to one declaration per line, and each line should be a complete declaration. When practical, variable names should be indented to the nearest logical tab after the widest type declaration. Multiple declarations should be aligned for readability.

```
Example:

DBAttributeName     associate_number;
UShort              error_count = 0;
PChar               home_address = NULL;
```

Note: Use of int is discouraged for portability except where needed by library functions. Use shorts instead of longs where possible for speed. Unless a variable needs to be signed use unsigned char, unsigned short, and unsigned long.

Simple data initialization:

For consistency, simple data initialization should occur during the declaration of variables as opposed to the executable section of code. This eliminates redundant entries of variable names and allows the executable code to be written (and read) as if all initializations have taken place automatically. This allows the reader to concentrate on the algorithm, not simple initializations.

```
Example:

PCharBuffer     input_buffer = NULL;
StringSize      string_length = 0;
Char            *buffer(NULL);
Int             i(0);
```

Global data:

All global data must be declared as "extern" in the interface file and with no scope qualifier in the source file that owns it. This will allow other modules that include the interface file to access the data but will only define storage for the data in the module that owns it.

Exactly one instance of the variable must occur without the "extern" qualifier, and that instance defines the storage.

Also, the theory of "one write, many read" should generally be applied to all global data. In other words, one module (the owner of the data) can read or write the data while all other modules may only read the data. This requires a module interface for functions gaining write access to other modules' data.

Pointers:

Pointers should be declared and used as "pointer to an object of type X." For example, you should not use a variable that is declared as "pointer to int" as a "pointer to char." Even with type casting, this may cause various difficulties in certain environments. If the object type being pointed to is not known, use a "void" pointer.

Example:

```
ConditionValue TransferData(
                const SizeT          number_bytes,
                const Void           *source_data,
                const SegmentType    *segment);
```

"const" and "volatile" qualifiers:

Be aware of the proper use of the "const" and "volatile" qualifiers for data declarations (a description of their use is in the ANSI standard). In general, use the "const" qualifier for data that doesn't change within the scope of the declaration, and use the "volatile" qualifier for data that gets changed by an action outside the scope of the declaration.

Examples:

```
ComCode OutputString (const char *string);

extern volatile unsigned long int ClockTicks;
```

Use of "const" allows the compiler to put constant data in sharable (read-only) memory. Use of "volatile" is absolutely mandatory in certain environments--the program may sometimes work without it, but is likely to be buggy. "volatile" tells the compiler not to do any optimization on this variable.

References:

References should be used when passing arguments which are larger than the size of a pointer. For example, you should not pass a structure to a function, instead use a variable that is declared as reference to that structure. If the object being referenced is not going to change, use the const modifier as well. References may be used for output parameters as well, with consistency being taking into consideration.

Example:

```
//   Data is a structure
MyObject::MyObject (
                const SizeT          number_bytes,
                const Data           &source_data,
                ReturnType           &error);
```

Static data and re-entrant code:

Be aware that the use of static data (data declared at the module level or within a function using the "static" qualifier) may make code nonre-entrant. There is no simple guideline to follow to ensure that this doesn't happen. However, if you have code that is used in a re-entrant fashion and the code has static data, it should be looked at as questionable code.

Symbolic Constants:

In general, constants should not be coded directly--especially if the same constant appears more than once in the code. Constants should have meaningful names assigned to them via either the "#define" preprocessor directive, const declaration or an enumeration declaration. This makes it much easier to administer changes in large and evolving programs because constants need only be changed in one place in the program.

Example:

```
#define MAXIMUM_NAME_LENGTH  20
#define MAXIMUM_NAME_SIZE (MAXIMUM_NAME_LENGTH+1)
const Int MAXIMUM_LINE_LENGTH 100
```

This way, if MAXIMUM_NAME_LENGTH changes in the future, only one declaration needs to be changed.

Don't redefine "NULL":

The pointer constant "NULL" is defined by the standard library (in "stddef.h", "stdio.h", "stdlib.h", "string.h", "locale.h", and "time.h") and should never be redefined by a program.

Structure, Union, and Enumeration Declarations

Overview:

The use of structures, unions, enumerations, and type declarations can significantly enhance a program's readability and maintainability. Each of these declarations have a similar format as shown below.

Structures, unions, and enumerations:

Structures, unions, and enumerations enhance the logical organization of code, offer consistent addressing of data elements, and can increase the efficiency and performance of some programs.

Format: Structures, unions, and enumerations should be formatted as follows:

```
typedef union
{
    InitialPacketType   initial_packet;
    TransferPacketType  transfer_packet;
    ...
} ProtocolPacket;

typedef enum
{
    START_STATE,
    IF_FOUND_STATE,
    ELSE_FOUND_STATE,
    ENDIF_FOUND_STATE
} ProcessingStates;

typedef struct
{
    Long            stack_size;
    StackElement    stack[MAXIMUM_STACK_SIZE];
} Stack;
```

If a structure needs to reference itself within it's declaration, use the 'tag' label prepended to the name of the structure. For example:

```
typedef struct TagStack
{
    Long            stack_size;
    StackElement    stack[MAXIMUM_STACK_SIZE];
    struct TagStack *ptr_stack;
} Stack;
```

Structure initialization can be done automatically by use of the explicit initializer. This insures that all members of a structure are set to defaults when instantiated.

```
Example:

typedef struct TagStack
{
     Long            stack_size;
     StackElement    stack[MAXIMUM_STACK_SIZE];
     TagStack(Void)
     {
          stack_size = 0;
     };
} Stack;
```

Type definitions and usage:

To ensure compatibility among different data types,

- always use explicit type casts where conversion is required
- use "typedef" declarations for all non-base data types

Format: typedefs should be formatted differently depending on whether they are used with or without a structure, union, or enumeration. In either case, format the typedef as you would format the type it is defining, but add the word "typedef" before the actual type.

```
Examples:

typedef PVoid DataPointer;

typedef struct
{
     PChar       macro_expansion;
     MacroTypes  macro_type;
} MacroDefinition;
```

Usage: Use "typedef" to declare a "struct" or a "union" as a type. Avoid declaring a tag for a "struct" or "union" unless the "struct" or "union" is self-referential or two or more "structs" or "unions" are mutually-referential.

```
Example:

typedef struct TagCommandType
{
     PChar                    command;
```

```
        struct TagCommandType     *next_command;
    } CommandType;
```

All other references to this structure use the "typedef" form of the name as follows:

```
CommandType *command_list;
```

General Source Formatting Guidelines

Overview:

There are a few general formatting guidelines that should be applied throughout your source code. A general description will be given first, followed by a detailed description for each language construct.

Indentation:

Indentation is defined in terms of logical tabs. A logical tab, or "soft tab," is four spaces.

All subordinate language constructs are to be indented one logical tab. Some examples of subordinate constructs are:

- fields within structures
- body of a loop
- body of an if statement

For assembly language a tab is eight spaces.

Line length:

To allow programs to be read and modified on a computer screen and allow space for listing and cross-reference utilities, no source line should exceed 80 characters in length.

Continuation lines:

When a statement needs to be continued on more than one line, the second line should be indented at least two logical tabs below the first line. The third and subsequent continuation lines should be indented even with the second continuation line.

Statements being continued onto multiple lines should be separated as follows:

- assignment statements and expressions - following an operator
- for statements - following a semicolon
- function parameters - following a comma or before the first parameter

Examples:

```
File_Stack[file_stack_size].file_name=
          Stream_Stack[current_stream].file_name;

status = ParseExplicitRule (
          expanded_line,
          target_list,
          dependency_list);
```

Page breaks:

A form feed should be inserted between major sections in a source file. At a minimum, a function definition that doesn't start on a page boundary should end on that same page. If it doesn't, use a form feed to make it start on a page boundary.

One statement per line:

Each line should contain only one statement. The only exception is the "else if" construct (where "else if (expression)" should appear on one line).

Compound statements:

A compound statement should be indented one logical tab from the surrounding code except for the braces.

Example:

```
if (string_length > 0)
{
    Total_Length += string_length;
    printf ("Total Length = %d\n", Total_Length);
}
```

"if" statement:

For a simple "if" statement with no "else" part, place the "if" statement and the conditionally executed statement on separate lines as follows:

```
if (today == MONDAY)
    number_of_mondays++;
```

If more than one statement is to be conditionally executed, use the following style:

```
if (*line_ptr != EOS)
{
   ReportError ("Extra input at end of line");
   return FAILURE;
}
```

"else" and "else if":

If an "else" clause is added to an "if" statement, use the following format:

```
if (today == MONDAY)
    number_of_mondays++;
else
    number_of_other_days++;
```

The "else if (expression)" should appear on a single line by itself and be formatted as follows:

```
if ( strcmp(today_buffer, "MONDAY") == 0 )
{
     number_of_mondays++;
     printf ("Number of Mondays is %d\n",
     number_of_mondays);
}
else if(strcmp(today_buffer, "TUESDAY")== 0)
{
     number_of_tuesdays++;
     printf ("Number of Tuesdays is %d\n",
     number_of_tuesdays);
}
else
    number_of_other_days++;
```

"return" statement:

Multiple returns in a function are discouraged. When they are necessary, care should be taken to insure any special clean up needed by the function is done before every return (e.g. closing a file opened by the function).

"switch" statement:

The body of a "switch" statement is enclosed in braces and indented one logical tab. Place each individual case clause on a new line. The statement associated with a case clause should be placed on the line following the case clause and should be indented two logical tabs from the "switch" statement. All statements following a case label should end with a "break" statement.

"default" case for "switch" statements:

All "switch" statements should have a "default" case which, if not expected, should signal some sort of serious error. The "default" case should always be the last case.

Falling through cases:

In general, avoid the characteristic of the "switch" statement that allows a block of code associated with one case label to fall through to the block of code associated with the next label. In those cases where it is unavoidable, make sure that it is well-documented.

However, you may associate several case labels with no intervening code with one block of code.

The following is an example of the "switch" statement:

```
switch (input_char)
{
    case 'A':
    case 'B':
       puts ("A or B pressed");
       break;

    case 'D':
       break;

    default:
       puts ("Wrong key pressed!");
       break;
}
```

Class definitions:

Start the class definition by listing the constructor(s) and destructor. List the destructor even if it doesn't do anything. Next, list **all** of the other public functions followed by the public data members. After the public class members are the protected members, and then the private members.

Following the class definition, typedef a pointer to the class. Name the pointer '*PClass*', where *Class* is the name of your class.

Example:

```
class MyClass : public Object
{
     public:
          MyClass(Void);
          ~MyClass(Void);

     protected:
          Int       Number_Of_Elements;

     private:
          Void      SetNumberOfElements(
                         const Int      number);

          Char FAR  *Element_List;
};
typedef MyClass FAR *PMyClass;
```

Arithmetic expressions:

Additional parentheses are recommended to clarify complex arithmetic expressions where operator precedence is not clear. In addition, some combinations of operators are hard enough to follow that separate statements should be used instead.

Example:

```
Instead of using...

    switch (*++string_ptr)

or even...

    switch (*(++string_ptr))
```

```
Consider using...

    string_ptr++;

    switch (*string_ptr)
```

Conditional expression operator:

The conditional expression operator is a combination of two operators: '?' and ':'. Use spaces on both sides of both operators. Avoid using the conditional operator except in macro definitions. When possible, use the "if" statement instead.

Sequence operator:

The sequence operator is a comma. Place a space after, but not before this operator. The use of the sequence operator should be avoided. There are certain exceptions where the sequence operator can be used within macros, but its use should be clearly justified, and there shouldn't be any other way to accomplish the same result.

General Source Code Guidelines and Recommendations

Assertions:

Assertions are a valuable tool for debugging an application. Assertions are used to verify that a condition is TRUE. If the condition is FALSE, then the OS will notify the developer of the file and line that the assertion failed on.

There are some general guidelines for when to use assertions. Parameters which are internal (ie not coming through a toolkit API) should be asserted. This verifies that the developer is not getting any data which is not expected. Assertions can also be used to verify that an object is in a particular state.

```
Example:

Void Myfunct(  const Char FAR      *buffer,
               const Int           size)
{
     assert(buffer != NULL);
     assert(size > 0);
     ...
}
```

Assertions will be compiled out by the compiler in an NDEBUG version.

Namespaces:

In the case where two or more interface files use the same name for different objects, namespaces should be used. The namespace should encompass that interface which is smaller in complexity. This ensures that the impact of the namespace is limited.

Constants vs defines:

Use of the const keyword instead of #define allows for the compiler to do type checking for the developer.

Enums vs defines:

Use of the enum keyword to describe groups of defines is a safer mechanism than define. By using enum, the compiler is able to do type checking for the developer.

Class scope enums vs constants and defines:

There are times when a class definition requires the use of a constant value which does not have to be available outside the scope of the class (ie to other modules which include this interface file). In these cases, the class can have a class scope enum which is used to represent this constant value. This ensures that the constant value is not defined within another interface file. In this case, the compiler may pick one of the two values and use that one. This may cause undesirable results.

```
Example:

class X
{
     ...
     enum {MAX_LENGTH = 10};

     Char       buffer[MAX_LENGTH];
};
```

Class scope types and structures:

There are times when a class definition requires the use of a structure which does not have to be available outside the scope of the class (ie to other modules which include this interface file). In these cases, the class can have a class scope strucure. This ensures that the structure is not defined within another interface file. This also limits the complexity of the public portion of the class definition by not exposing unneccesary types to the developer.

```
Example:

class X
{
     ...
     typedef struct
     {
          float     real_value;
          float     imaginary_value;
     } ImaginaryNumber;

     ImaginaryNumber     Value;
};
```

Inline and templates vs macros:

Use of inline functions for commonly used macros insures that the return type is the type expected. Take this example:

```
#define max(a,b) (a > b ? a : b)

float      x = 1.9;
int        y = 1;
int        z;

z = max(x,y);   // z is 1
```

An inline function taking integers would have flagged this with a warning. The inline function guarantees that the input and output are the expected type.

```
inline int max(int a, int b)
     {return (a > b ? a : b);}
```

However, if the macro is expected to support more than one type, then a template function would be the best alternative.

Local scoping of variables:

In the case where a variable is used in a very limited scope, that variable may be defined in that scope. This moves variable declarations closer to the location where they are used. Care must be taken to insure that variable names do not clash.

```
Example:

if (i == 100)
{
     String    string;
     ...
}
```

General Toolkit API and Header Guidelines

C Programming Interface

The toolkit should export a C programming interface. The header file should not be dependent upon any C++ language extensions. It should contain only C language elements.

Automatic extern C for C++ code

```
Example:
#ifdef __cplusplus
extern "C" {
#endif
      .
      .
      .
#ifdef __cplusplus
}
#endif
```

Development Environment

No compiler dependencies

The code should not be dependent upon any particular compiler. Although the code may be written using only one compiler, it should be designed and tested to work with all equivalent compilers in the target environment. #ifdef's may be necessary and are allowed to remove the dependency on particular compilers. The currently supported list of compilers should be available through the project leaders and are not included here due to the frequent change in versions.

Out of the box compiling on target platform

The toolkit should be set to compile on the target platform without having to set any special defines upon installation. _Windows is predefined for all Windows compilers.

Header file nesting

Header files from the various toolkits should try not to include header files from other lower level toolkits unless absolutely necessary. No extraneous header files should be included.

Use Platform Native Data Types

The toolkit should accept the platform native data types at the API level, either in structures or as parameters.

For performance reasons

Since the toolkit needs to use the native data types (especially Microsoft Windows bitmaps and device contexts) anyway, why force a conversion into a DataBeam type just to convert it back again. Data being communicated from remote systems will be converted into a platform native type immediately.

For convenience

It is easiest for developers to give the toolkit the same data types that they are using on the platform.

Naming Conventions

While there may never be total agreement as to what is "the best" style to use for naming conventions, it is necessary to agree on "an acceptable" style that will be consistent across the API. Consistency and clarity wil help make this product easier to understand and use. The following conventions should be used for the API.

Name Prefixes

To avoid type name clashes, every function, data type, structure, constant, etc. for this product should be prefixed with a short abbreviation of the name of the tookit. Note that it is not necessary to prefix the names of the members in a structure, or the names given to parameters to functions. Examples of prefixes: FTAT will have an "FT" prefix. GCCAT has a "GCC" prefix. SWAT will be prefixed with the letters "SW", etc.

```
Examples:

SW_FILL_PATTERN_HORIZONTAL constant
SWError data type
SWObjectProperties data structure
SWCreateView() function
```

Use DataBeam coding standards naming style

Functions and data types should be named as described in the DataBeam *Programmer's Coding Guide*. To briefly summarize (for more information, see the guide):

Booleans

Parameters or variables that are Boolean types should contain a to-be helping verb and should indicate the TRUE state.

```
Examples:
     BOOL       capabilities_were_added;
     BOOL       is_actively_enrolled;
     BOOL       conference_is_conducted;
```

Pointers

Parameters or variables that are pointers should have descriptive names and the * should be shown with the identifier. Basically, do not use the DataBeam style of creating P data types. The only exception to this rule is the well know string type which includes a pointer to a data type.

```
Examples:

Good:
SWError APIEntry SWGetProperties(
          SWObject              object_handle,
          SWProperties   FAR *object_properties);

Bad:
SWError APIEntry SWGetProperties(
          SWObject              object_handle,
          PSWProperties       object_properties);
```

Strings

As mentioned above, strings and string-like types may include a pointer. When creating a string type, always create a base type as well.

```
Example:

typedef unsigned short          SWUnicodeCharacter;
typedef SWUnicodeCharacter FAR *SWUnicodeString;
```

Unions

Unions should be named with just a "u".

```
Example:

typedef struct
{
    GCCCapabilityIDType  capability_id_type;
    union
    {
        unsigned short   standard_capability;
```

```
        GCCObjectKey      non_standard_capability;
    } u;
} GCCCapabilityID;
```

Consistency

The API should be consistent is style and naming conventions. Try to name functionally identical parameters indentically. For example, if bitmaps and workspaces and views all have a boundary size, that boundary size should have the same name for each, whether that is called bounding_box, bounding_rectangle, bounding_region, boundry_area, or clipping_region, they should all have the same name.

Function Parameter Considerations

Parameter Ordering

As dictated by the coding standard, function parameters should be listed in the order of input parameters, followed by input-output parameters, followed by output parameters. For functions that have a "parent handle", it should be the first of the input parameters.

Generally, the parameters should be grouped by importance (if it can be determined). The order for each (in, in/out, and out) will be most important parameters, followed by less important parameters, followed by the least important parameters. Usually, the most important parameters will be directly related to the name of the function. For example, a function that creates a workspace will have the workspace handle as the most important out-parameter.

Passing Lists to or Receiving Lists from Functions

For consistency, any function that needs to pass or return list and number of elements in the list parameters should pass the number of elements parameter first, followed by the list. This rule should also be followed in structures.

Example:

```
void ListFunction(
          short                 number_in_list,
          item FAR * FAR        *list_of_items);
```

Function Return Types

Every function's return type should be an error code. If a function needs to return data, use pointers. This allows the API to indicate successful completion or errors

for every function. The error return type should probably be implemented as an enum which contains all the possible error values.

Enums instead of defines

Due to type checking benefits, enums are preferred over defines. The ANSI C style which specifies the enums exactly is probably best.

```
Example:

typedef enum
{
     GCC_NO_ERROR                          = 0,
     GCC_NOT_INITIALIZED                   = 1,
     GCC_ALREADY_INITIALIZED               = 2,
} GCCError;
```

For errors, results, etc. that are the same between toolkits, they should be named the same.

Appendix A: Listing of "databeam.h"

This listing is provided as an example only. As of the date of this printing, this was the current databeam.h. However, due to changes, this file may have been updated. For the most up-to-date version of the file, the project leader will be able to retrieve it from Source Safe. (Currently it is also located in k:\source\codestd.)

```
/*
 * databeam.h
 *
 *      Copyright (c) 1993 - 1995 by DataBeam Corporation, Lexington, KY
 *
 * Abstract:
 *      This file defines common extensions to the C++ language for
 *      use at DataBeam Corporation.
 *
 * Author:
 *      James P. Galvin, Jr.
 *      Brian L. Pulito
 *      Carolyn J. Holmes
 *      John B. O'Nan
 *
 * Revision History
 *      08AUG94             blp             Added UniChar
 *      15JUL94             blp             Added Strcmp
 */

#ifndef _DATABEAM_
#define _DATABEAM_

#ifdef  _WINDOWS
#       include <stdio.h>
#       include <stdlib.h>
#       include <windows.h>
#       include <windowsx.h>
#ifdef __BORLANDC__
#       include <mem.h>
#else
#       include <string.h>
#endif
#else
#       include <stdio.h>
#       include <stdlib.h>
#       include <string.h>
#endif

/*
 *      These macros resolve the FAR and HUGE types used in the typedefs below.  In
 *      a windows enviroment, these will have a valid value.  In other environments,
 *      they will probably be NULL.
 */
#ifdef  _WINDOWS
#       ifdef    WIN32
```

```
#               ifndef  FAR
#                       define  FAR
#               endif
#               ifndef  HUGE
#                       define  HUGE
#               endif
#       else
#               ifdef   FAR
#                       ifdef   __SC__
#                               define  FAR             __far
#                       endif
#               else
#                       define  FAR                     _far
#               endif
#               ifdef   HUGE
#                       ifdef   __SC__
#                               define  HUGE            __huge
#                       endif
#               else
#                       define  HUGE                    _huge
#               endif
#       endif
#else
#       define  FAR
#       define  HUGE
#endif

/*
 * The following two macros can be used to get the minimum or the maximum
 * of two numbers.
 */
#ifndef min
#       define  min(a,b)(((a) < (b)) ? (a) : (b))
#endif

#ifndef max
#       define  max(a,b)        (((a) > (b)) ? (a) : (b))
#endif

/*
 *      These macros define all the use of standard functions that are available
 *      in some form in all environments.
 */
#ifdef  _WINDOWS
#       ifdef   __DLL__
#               define  Malloc(size)            (GlobalAllocPtr(GHND | GMEM_DDESHARE, size))
#       else
#               define  Malloc(size)            (GlobalAllocPtr(GHND, size))
#       endif
#       define  ReAlloc(gmem_ptr, size)         (GlobalReAllocPtr(gmem_ptr, size, GHND))
#       define  Free(ptr)                       (GlobalFreePtr(ptr))
#       define  Memcmp(dest, src, size)         (_fmemcmp(dest, src, size))
#       define  Memcpy(dest, src, size)         (_fmemcpy(dest, src, size))
```

```
#       define  HMemcpy(dest, src, size)        (hmemcpy(dest, src, size))
#       define  Memset(dest, c, size)           (_fmemset(dest, c, size))
#       define  Strcpy(dest, src)               (lstrcpy(dest, src))
#       define  Strncpy(dest, src, size)        (lstrcpyn(dest, src, size))
#       define  Strcat(dest, src)               (lstrcat(dest, src))
#       define  Strlen(dest)                    (lstrlen(dest))
#       define  Strcmp(src1,src2)               (lstrcmp(src1, src2))
#else
#       define  Malloc(size)                    (malloc(size))
#       define  ReAlloc(ptr, size)              (realloc(ptr, size))
#       define  Free(ptr)                       (free(ptr))
#       define  Memcmp(dest, src, size)         (memcmp(dest, src, size))
#       define  Memcpy(dest, src, size)         (memcpy(dest, src, size))
#       define  HMemcpy(dest, src, size)        (memcpy(dest, src, size))
#       define  Memset(dest, c, size)           (memset(dest, c, size))
#       define  Strcpy(dest, src)               (strcpy(dest, src))
#       define  Strncpy(dest, src, size)        (strncpy(dest, src, size))
#       define  Strcat(dest, src)               (strcat(dest, src))
#       define  Strlen(dest)                    (strlen(dest))
#       define  Strcmp(src1,src2)               (strcmp(src1, src2))
#endif


/*
 *      This typedef defines Boolean as an int, rather than an enum.  The
 *      thinking is that this is more likely to be compatible with other
 *      uses of Boolean (if any), as well as with the use of "#define" to
 *      define TRUE and FALSE.
 */
#ifndef DBBoolean
typedef int                                     DBBoolean;
typedef int *                                   PDBBoolean;
typedef const int *                             PCDBBoolean;
typedef int FAR *                               FPDBBoolean;
typedef const int FAR *                         FPCDBBoolean;
typedef int HUGE *                              HPDBBoolean;
typedef const int HUGE *                        HPCDBBoolean;
#endif


/*
 *      These defines set up values that would typically be used in conjunction
 *      with the type Boolean as defined above.
 */
#ifndef FALSE
#       define  FALSE 0
#endif
#ifndef TRUE
#       define  TRUE  1
#endif
#ifndef NO
#       define  NO    0
#endif
```

```
#ifndef  YES
#        define   YES     1
#endif
#ifndef  OFF
#        define   OFF     0
#endif
#ifndef  ON
#        define   ON      1
#endif


/*
 * EOS can be used for the NUL byte at the end of a string.  Do not
 * confuse this with the pointer constant "NULL".
 */
#define EOS     '\0'


/*
 *      The following is a list of the standard typedefs that will be used
 *      in all programs written at DataBeam.  Use of this list gives us full
 *      control over types for portability.  It also gives us a standard
 *      naming convention for all types.
 */
typedef  char                                   Char;
typedef  unsigned char                          UChar;
typedef  char *                                 PChar;
typedef  const char *                           PCChar;
typedef  unsigned char *                        PUChar;
typedef  const unsigned char *                  PCUChar;
typedef  char FAR *                             FPChar;
typedef  const char FAR *                       FPCChar;
typedef  unsigned char FAR *                    FPUChar;
typedef  const unsigned char FAR *              FPCUChar;
typedef  char HUGE *                            HPChar;
typedef  const char HUGE *                      HPCChar;
typedef  unsigned char HUGE *                   HPUChar;
typedef  const unsigned char HUGE *             HPCUChar;

typedef  short                                  Short;
typedef  unsigned short                         UShort;
typedef  short *                                PShort;
typedef  const short *                          PCShort;
typedef  unsigned short *                       PUShort;
typedef  const unsigned short *                 PCUShort;
typedef  short FAR *                            FPShort;
typedef  const short FAR *                      FPCShort;
typedef  unsigned short FAR *                   FPUShort;
typedef  const unsigned short FAR *             FPCUShort;
typedef  short HUGE *                           HPShort;
typedef  const short HUGE *                     HPCShort;
typedef  unsigned short HUGE *                  HPUShort;
typedef  const unsigned short HUGE *            HPCUShort;
```

```
typedef  int                          Int;
typedef  unsigned int                 UInt;
typedef  int *                        PInt;
typedef  const int *                  PCInt;
typedef  unsigned int *               PUInt;
typedef  const unsigned int *         PCUInt;
typedef  int FAR *                    FPInt;
typedef  const int FAR *              FPCInt;
typedef  unsigned int FAR *           FPUInt;
typedef  const unsigned int FAR *     FPCUInt;
typedef  int HUGE *                   HPInt;
typedef  const int HUGE *             HPCInt;
typedef  unsigned int HUGE *          HPUInt;
typedef  const unsigned int HUGE *    HPCUInt;

typedef  long                         Long;
typedef  unsigned long                ULong;
typedef  long *                       PLong;
typedef  const long *                 PCLong;
typedef  unsigned long *              PULong;
typedef  const unsigned long *        PCULong;
typedef  long FAR *                   FPLong;
typedef  const long FAR *             FPCLong;
typedef  unsigned long FAR *          FPULong;
typedef  const unsigned long FAR *    FPCULong;
typedef  long HUGE *                  HPLong;
typedef  const long HUGE *            HPCLong;
typedef  unsigned long HUGE *         HPULong;
typedef  const unsigned long HUGE *   HPCULong;

typedef  float                        Float;
typedef  float *                      PFloat;
typedef  const float *                PCFloat;
typedef  float FAR *                  FPFloat;
typedef  const float FAR *            FPCFloat;
typedef  float HUGE *                 HPFloat;
typedef  const float HUGE *           HPCFloat;

typedef  double                       Double;
typedef  double *                     PDouble;
typedef  const double *               PCDouble;
typedef  double FAR *                 FPDouble;
typedef  const double FAR *           FPCDouble;
typedef  double HUGE *                HPDouble;
typedef  const double HUGE *          HPCDouble;

typedef  long double                  LDouble;
typedef  long double *                PLDouble;
typedef  const long double *          PCLDouble;
typedef  long double FAR *            FPLDouble;
typedef  const long double FAR *      FPCLDouble;
typedef  long double HUGE *           HPLDouble;
```

```
typedef  const long double HUGE *              HPCLDouble;

typedef  void                                  Void;
typedef  void *                                PVoid;
typedef  const void *                          PCVoid;
typedef  void FAR *                            FPVoid;
typedef  const void FAR *                      FPCVoid;
typedef  void HUGE *                           HPVoid;
typedef  const void HUGE *                     HPCVoid;

/*
 *      Temporary fix for compatibility with the Symantec compiler, which doesn't
 *      recognize wchar_t as a valid type.
 */
typedef  unsigned short                        UniChar;
typedef  UniChar        *                      PUniChar;
typedef  UniCharFAR     *                      FPUniChar;

#endif
```

Appendix B: Interface File Template

```
/*
 * Copyright (c) 1996 by DataBeam Corporation, Lexington, KY
 *
 * Abstract:
 *
 * Protected Member Variables:
 *
 * Portable:
 *
 * Caveats:
 *
 * Author:
 *
 */
```

In the event that there are global variables which are declared in the implementation file, then the Variables section should be added.

```
#ifndef _?_H_
#define _?_H_

/*
* External Interfaces
 */
```

All includes necessary to compile with this interface file included go here.

```
/*
 * Constants, structures, and typedefs
 */
```

Defines, macros, constants, enums and typedef'ed structures/unions which need to be globally available to modules which include this interface file should be defined here.

```
/*
 * External data
 */
```

Any data declared in the implementation file which is global (ie used in other files) should be externed here.

```
/*
 * Function prototypes
 */
```

Any functions declared in the implementation file which can be used in other implementation files should be prototyped here.

```
/*
 * Class definition
 */
```

Only public and protected member functions need to have the full function header in the interface file. Also, any prototyped "C" functions or C++ methods should be described.

```
/*
 * Function Name:
 *
 * Public (or Protected)Function Description:
 *
 * Formal Parameters:
 *
 * Return Value:
 *
 * Side Effects:
 *
 * Caveats:
 *
 */
```

Each of the formal parameters specifies (i), (o), or (i/o) depending on whether it is an input only parameter, an output only parameter, or an input/output parameter. A simple description of the parameter should follow.

Each of the headings in the block comment should be listed, even if the response is NONE.

Add the *Implicit Parameter* heading to the comment block if the function uses global variables. List the global variables following the heading in the same format as the *Formal Parameters*. Global variables should be avoided.

Note: We do not list class instance variables as *Implicit Parameters.*

```
#endif
```

Appendix C: Implementation File Template

```
/*
 * Name of file:
 *
 * Copyright (c) 1996 by DataBeam Corporation, Lexington, KY
 *
 * Abstract:
 *
 * Private Instance Variables:
 *
 * Portable:
 *
 * Caveats:
 *
 * Author:
 *
 */
```

In the event that there are global variables which are declared in the implementation file, then the Variables section should be added.

```
/*
 * External Interfaces
 */
#include "databeam.h"
#include "?.h"

/*
 * Constants, structures, and typedefs
 */
```

Defines, macros, constants, enums and typedef'ed structures/unions which do not need to be globally available should be defined here.

```
/*
 * Global data
 */
```

Globally defined data should go here.

```
/*
 * Static data
 */
```

Data which is local to this module should go here.

Functions which are already fully described in the interface file should use an abbreviated form of the function header as follows:

```
/*
 * Function Name:
 *
 * Public (or Protected) Function Description
 */
```

Functions which are not fully described in the interface file should use the full form of the function header as follows:

```
/*
 * Function Name:
 *
 * (Private) Function Description:
 *
 * Abstract:
 *
 * Formal Parameters
 *
 * Return Value:
 *
 * Side Effects:
 *
 * Caveats:
 *
 */
```

Each of the formal parameters specifies (i), (o), or (i/o) depending on whether it is an input only parameter, an output only parameter, or an input/output parameter. A simple description of the parameter should follow.

Each of the headings in the block comment should be listed, even if the response is NONE.

Add the *Implicit Parameter* heading to the comment block if the function uses global variables. List the global variables following the heading in the same format as the *Formal Parameters*. Global variables should be avoided.

Note: We do not list class instance variables as *Implicit Parameters.*

Appendix D: Accepted Abbreviations

Listed below are the accepted abbreviations in alphabetic order for use at DataBeam. This list may be expanded at a future time.

Char	Character
Com	Communication
Ptr	Pointer
Temp	Temporary