SAE The Engineering Society For Advancing Mobility Land Sea Air and Space®
INTERNATIONAL
400 Commonwealth Drive, Warrendale, PA 15096-0001

AEROSPACE RECOMMENDED PRACTICE

SAE ARP4868

Issued 2001-10

Application Programming Interface Requirements for the Presentation of Gas Turbine Engine Performance on Digital Computers

TABLE OF CONTENTS

TO PLACE A DOCUMENT ORDER: (724) 776-4970 FAX: (724) 776-0790 SAE WEB ADDRESS: http://www.sae.org

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

1. SCOPE:

This SAE Aerospace Recommended Practice (ARP) specifies a set of functions and their expected behavior that constitute an Application Programming Interface (API) for gas turbine engine customer programs. The main body of this document contains a description of each of the API function calls and of the data that passes through these functions. Implementations of this API in specific programming languages are contained in separate appendices. These appendices include language specific details and the definitions for each function as required for the given language.

AS681 is the parent document of ARP4868. All of AS681 applies to customer engine programs written to conform to this document.

1.1 Purpose:

The function call based API specified in this ARP represents an alternative to the FORTRAN COMMON block structure historically used to communicate with an engine program. The COMMON block transport mechanism for communications with gas turbine engine customer programs is still supported. ARP4191 is a companion document. It contains the specifications for using COMMON blocks as the basis of an API to communicate with an engine program. It represents an alternative to meeting the requirements specified in AS681.

A function call API allows the calling program to directly reference an engine parameter by name in the function call list. Only those parameters that are available from the given engine program need be present in the list of variables. There is no need to include variables simply to maintain alignment in predefined COMMON blocks. The functions in the API are not restricted to setting and retrieving the numerical value of the engine program parameters. There are documentation type functions that list the engine parameters available to the user and that give information about each parameter. A function call based API also provides an opportunity to perform parameter specific calculations such as calculating infrequently used parameters only when the calling program specifically requests the value, so called calculate-on-demand. Lastly, a function call API, either as stand-alone functions or as methods of an object in an object-oriented program, integrates very well with object-broker, distributed processing and object-oriented methods.

A COMMON block API does have a speed advantage, since directly sharing memory represents the fastest way to communicate between programs. However, such an interface does have several shortcomings. A COMMON block API requires the calling program and the engine program to map the same parameter in the same location in the COMMON block definition in each program. Misalignment of the COMMONs will result in miscommunication of data across the interface. Engine programs have also not been able to be self-documenting, other than a brief identifier, since only limited character data can be communicated through the existing COMMONs. Lastly, COMMON block APIs do not integrate well with emerging technologies such as Common Object Request Broker Architecture (CORBA) and other distributed processing and object-oriented methods.

2. REFERENCES:

2.1 Applicable Documents:

The following publications form a part of this document to the extent specified herein. The latest issue of SAE publications shall apply. The applicable issue of other publications shall be the issue in effect on the date of the purchase order. In the event of conflict between the text of this document and references cited herein, the text of this document takes precedence. Nothing in this document, however, supersedes applicable laws and regulations unless a specific exemption has been obtained.

2.1.1 SAE Publications: Available from SAE, 400 Commonwealth Drive, Warrendale, PA 15096-0001.

AS681	Gas Turbine Engine Steady-State and Transient Performance Presentation for Digital Computer Programs
AS755	Aircraft Propulsion System Performance Station Designation and Nomenclature
ARP4191	Gas Turbine Engine Performance Presentation for Digital Computer Programs Using FORTRAN 77

2.2 Definitions:

ENGINE PROGRAM: Gas turbine engine performance program for digital computers.

API: Application Programming Interface: The formalized mechanism for communication between the calling program and the engine program.

PARAMETER NAME: The final element of a path name that refers to a variable within the engine program. Example: T41 or ALT.

CALCULATED PARAMETER: Any engine program parameter with a value calculated by the engine program. Often, these are referred to as output parameters. However, since the value of the input parameters may also be output through this API, the term "calculated" is used to avoid ambiguity.

INPUT PARAMETER: A parameter that can be set by the customer and which the engine program will not overwrite.

CATEGORY: A collection of elements, either engine program parameters or subcategories. All categories may contain subcategories. The result is a tree-like structure with all categories branching from a single root category. This structure has many similarities to the file structure of a computer operating system, where a category is analogous to a directory and a parameter to a single file.

ROOT CATEGORY: The base level category that contains, directly or indirectly, all subcategories and parameters in the engine program. A path name beginning with a leading backslash indicates that the path begins with the root category. The root category is explicitly referenced by a character string containing only the backslash character. An empty string does not reference the root category.

2.2 (Continued):

DEFAULT CATEGORY: A partial path name starting at the root category that defines the starting point for all relative path names. If the default category is an empty string, all path names are full path names.

SUBCATEGORY: A category that is contained within another category. With the exception of the root category, each category is a subcategory.

ELEMENT: A generic term for categories and parameters. The term element is used when referring to the contents of a category, which may contain both parameters and subcategories.

DIRECT ACCESS IDENTIFIER: A globally unique non-zero integer associated with each element. It provides a way to directly reference each element. An ID value of zero has special meaning within the API functions, and must not be used as the ID of any element. If direct access identifiers are not utilized then the ID value for all elements must be zero. If they are utilized, then none may be zero.

DATA TYPE: An integer value associated with each engine program element that identifies that element as being an integer, single precision floating point, double precision floating point or character parameter; or a category. The data type also indicates whether the parameter represents a single value or a table of values.

HIERARCHY: The organization of categories and parameters within the root category. More than one hierarchy may be defined for a given engine program, though only one may be active in a single execution session.

PATH NAME: A string that contains category names and possibly a parameter name. Each element name in the path name is separated by a backslash character. This string describes a path through the category hierarchy. Read from left to right, the path descends from category to subcategories to parameters. If the path contains a parameter name, the parameter name must be the last element of the path.

FULL PATH NAME: A path name that begins at the root level category (i.e. begins with a backslash). Only full path names are guaranteed to be unique. Individual element names and partial path names may reappear in other branches of the hierarchy.

RELATIVE PATH NAME: A path name, which when appended to the path name of the default category, results in a full path name.

KEYWORD: A unique character string associated with a hierarchy. More than one keyword may be associated with each hierarchy.

EMPTY STRING: An empty string contains either an initial ASCII NULL character or all blank characters.

2.2 (Continued):

SUPPLIER: The organization or person that delivered or created the engine program.

CUSTOMER: The person that executes the engine program through the API function calls.

INTEGER: A whole number of at least 32 bits unless otherwise agreed to by the customer and supplier. The supplier should document the exact size of integers in the engine program. 32 bits is specified because the largest positive 16 bit signed integer is 32,768. This is not sufficiently large to support some possible ID number schemes.

CORBA: Common Object Request Broker Architecture. A specification defining the inter-process communication between two programs using Internet protocols.

3. OVERVIEW:

Current gas turbine engine programs use a FORTRAN-based API that relies on COMMON blocks for data communication. The use of COMMON blocks results in a position dependent API. Misalignment of the COMMON block definitions in the engine program and in the calling program will result in miscommunication of data across the interface. The use of numerical COMMON blocks as the only communication channel between the engine program and the customer does not allow the engine program to be self-documenting. COMMON block APIs do not integrate well with emerging technologies such as CORBA and other distributed processing and object-oriented methods that rely on function and method calls as their fundamental communication mechanism.

This document defines an alternative, position-independent, function-based API for engine programs. A function-based API provides a method to reference parameters directly by name, thus eliminating the need to align parameter locations in named COMMON blocks. Some of the API functions send and return numerical values. Others provide access to documentation that may be contained within the engine program. This documentation can include descriptions of the engine program, as well as information about the engine program parameters, such as descriptions, input/output status, and associated units definitions.

The supplier includes the API functions as an integral part of the engine program. This document specifies only those aspects and behaviors of the API functions that are visible to the calling program. Suppliers are free to determine how they implement the API functions. All interactions between the customer and the engine program should be through the API functions. The engine program supplier, with the prior agreement of the customer, is allowed to include additional API functions.

3.1 Multiple Programming Language Support:

The language in which the API functions are implemented should be agreed to by the customer and supplier. It is possible, but not recommended, for API functions written in one programming language to be called by a program written in another language. There are numerous inter-language issues that have to be addressed when the API and calling programs are written in different languages. This document defines separate sets of functions for each programming language. The main body of this document specifies the functionality expected of the API functions as well as common aspects of the structure and content of the data passing through those functions. The appendices, starting with Appendix C, contain the definitions of the API functions as they would be expressed in a specific programming language. Any language-specific issues, such as string termination and array indexing, are addressed in the applicable appendix. The function names, the number of arguments and the argument names vary between different programming languages.

3.2 Parameter Organization:

This API supports the concept of a parameter hierarchy that is analogous to the file structure of a computer operating system. Just as a file system is made up of directories that contain subdirectories and files, a parameter hierarchy may be made up of categories that contain subcategories and parameters. This capability is included to provide an organization and structure to the engine program parameters. The categories and subcategories also provide a context that is intended to help the customer understand the purpose and meaning of the parameters within the engine program. For example, a category named "BleedFlows" should clearly tell the user where to look for parameters related to the bleed flows in the engine program.

All conforming engine programs will have, as a minimum, a single category referred to as the root category. The root category, and any category in general, may contain subcategories as well as parameters. The result is a highly flexible parameter hierarchy. An engine program may support multiple parameter hierarchies. See Appendix A for an example of two possible hierarchy structures that a given engine program might support. The listHierarchies function provides a listing of the hierarchies available for the engine program. The user can select which hierarchy to use at run-time. Only one hierarchy may be used in each execution session.

The customer may query the engine program for a list of the elements within a given category. Starting the query with the root category and querying each subcategory in turn will result in a listing of the entire hierarchy. Thus, the hierarchy can be extracted from the engine program at run-time. The calling program does not need to be preprogrammed with the hierarchy structures in order to work with the engine program.

3.3 Element Attributes:

The three required attributes for parameters or categories are name, data type, and parameter I/O status. For parameters, the optional attributes are a direct access identifier, a description string, a security classification string, and an engineering units string. For categories, the optional attributes are a direct access identifier, a description string, and a security classification string.

3.4 Typical Order of Calls to the API Functions:

The following represents a typical order of events for communication between the calling program and the engine program. The name in courier font is the API function responsible for performing the action in the given step. The required or optional notation for each step refers to whether that step is required for successful execution. The initializeProgram function must be called before any of the remaining functions except programInfo and listHierarchies. The order of the calls to the other functions is not limited to what is given below, but represents a typical sequence.

1. Get information about the engine program – programInfo [Required]
2. List the available parameter hierarchies – listHierarchies [Optional]
3. Initialize the engine program interface – initializeProgram [Required]
4. List category and parameter elements in the engine program – listCategory [Optional]
5. Get additional descriptive information for each element – getDescription [Optional]
6. Get the current default category – getDefaultCategory [Optional]
7. Change the default category – setDefaultCategory [Optional]
8. Specify input values for a given case – setValue [Required]
9. Get a list of parameters that have changed I/O status – listChangedIOstatus [Optional]
10. Execute the engine program and examine returned Numerical Status Indicators (NSIs) – runEngProgram [Required]
11. Retrieve values for desired parameters – getValue [Required]
12. Retrieve a list of values at or beyond their limits – listLimitParameters [Optional]
13. Reset the input parameters to the supplier defined baseline values – resetEngProgram [Optional]
14. Repeat steps 8 through 13 until all the desired cases have been executed.

4. PROGRAM DOCUMENTATION:

4.1 User Documentation:

The program supplier will supply all the documentation specified in AS681. The following information that is unique to engine programs conforming to ARP4868 should be included with the user documentation:

1. The specific programming language used to implement the API functions.
2. A list of all parameter hierarchies available.
3. A map showing the structure of each of these hierarchies. This map is to include all the available parameters.
4. The maximum size of fixed size character string and array parameters or character string attributes, if applicable.
5. A list of security classification strings and their relative ranking.

5. ELEMENT ATTRIBUTES:

This API specifies a number of attributes for each element. The following sections list these attributes. The engine program supplier must provide those attributes marked as required for each element in the engine program. The optional attributes are not required for proper execution of the engine program, though it is highly recommended that the supplier include these attributes for each element. See Table 1 for a summary of the required and optional attributes for categories and parameters.

TABLE 1 - Required and Optional Attributes

	Name	ID	I/O Status	Data Type	Security Classification	Description	Units
Parameters	Req	Opt	Req	Req	Opt	Opt	Opt
Categories	Req	Opt	Req	Req	Opt	Opt	N/A

The API functions listed after the description of each attribute are those that contain that attribute in the argument list.

5.1 Path Name:

Each full path name in the engine program must be unique. The individual element names contained within the path are not required to be globally unique. A backslash is used to separate the element names. The individual element names may contain only alphanumeric, period, and underscore characters. They may not begin with a number. They also may not be empty or null strings.

Path name is output of listCategory, listChangedIOstatus, getDefaultCategory, listLimitParameters. Path name is input to getDescription, setValue, getValue, setDefaultCategory.

5.2 Direct Access ID:

The direct access ID attribute is a globally unique, non-zero integer associated with each element in the engine program. The direct access ID is an optional attribute. However, partial implementation of direct access IDs is not allowed. If direct access IDs are implemented, an ID value must be given for all elements. If direct access IDs are not implemented, then the ID value for all elements must be set to zero.

The purpose of the ID is to provide a mechanism for fast element access. The uniqueness of each ID within the engine program allows the engine program to directly locate the desired parameter. Location by name, on the other hand, requires a potentially large number of string comparisons to match the full path name.

5.2 (Continued):

The supplier does not have to follow any set numbering scheme when assigning IDs to elements. The only requirement is that there be a one-to-one correspondence between each ID number and one specific engine program element. Further, the IDs are not guaranteed to be the same for each execution of the engine program. For example, the IDs might represent memory addresses that will be different each time the engine program is loaded into memory. The user should therefore use the appropriate API function to get the IDs each time the engine program is run.

A distinction is made between category IDs and parameter IDs for some functions. This is because some API functions require an ID that is associated with a category and an ID that is associated with a parameter. Thus, separate argument names are required. However, both category IDs and parameter IDs belong to the same set of integer values. Thus, there are no duplicate integer values in the union of the set of category IDs and parameter IDs.

Direct access ID is output of listCategory, getDescription, setDefaultCategory, setValue, getValue.
Direct access ID is input to listCategory, getDescription, listChangedIOstatus,setDefaultCategory, getDefaultCategory, setValue, getValue, listLimitParameters.

5.3 Element I/O Status:

An element can have the following I/O attributes:

- active The parameter is in use.
- inactive The parameter is not in use for a particular engine program set-up.
- input The parameter can be set by the user but not by the engine program. The program may provide a default value.
- calculated The parameter is calculated by the engine program and should not be set by the user.
- category The parameter is a category. Even though I/O status does not apply to a category, a value must still be given.

(Note: “output”, in this document, is not an element I/O attribute but only a direction of information flow)

The integer values for element I/O status are:

0 = Category
1 = Input/active
2 = Calculated/active
3 = Input/Inactive
4 = Calculated/Inactive

5.3 (Continued):

Changing the value of one input parameter may result in changing the I/O status of other parameters. The user can call the listChangedIOstatus function to get a list of parameters with an I/O status different than the last time that function had been called.

Element I/O status is output of listCategory, getDescription, listChangedIOstatus.
Element I/O status is not input to any API function.

5.4 Data Type:

The data type attribute is used to indicate if the element is a category or a numerical or character string parameter. For parameters, it also indicates whether the parameter is a single value, a simple array of values, or a packed array of values. See section 7 for more information about packed arrays.

The data type indicated by each value is shown below. A value of 0 indicates that the element is a category. Values from 1 to 4 indicate that the element is a parameter containing a single value. Values from 11 to 14 indicate that the element is a parameter containing a simple array data. Values from 21 to 24 indicate that the element is a parameter containing a 2-D or higher table of data packed into a 1-D array.

The integer values for data type are:

0 = Category
1 = Integer Number
2 = Single Precision Floating Point Number
3 = Double Precision Floating Point Number
4 = Character String
11 = Array of Integer Numbers
12 = Array of Single Precision Floating Point Numbers
13 = Array of Double Precision Floating Point Numbers
14 = Array of Character Strings
21 = Table of Integer Numbers
22 = Table of Single Precision Floating Point Numbers
23 = Table of Double Precision Floating Point Numbers
24 = Table of Character Strings

Data type is output of listCategory, getDescription.
Data type is not the input of any API function.

5.5 Security Classification:

This attribute is a character string that indicates the security classification of an element. Examples of the security classification attribute include: UNCLASSIFIED, PROPRIETARY, COMPANY CONFIDENTIAL, and SPECIAL ACCESS REQUIRED. The security classification attribute is assigned by the supplier in accordance with governing security guidelines. The attribute is provided to assist the customer in handling output from the engine program in the case where different levels of classification are applied to some parameters.

The security classification is output of listCategory, getDescription.
The security classification is not input to any API function.

5.6 Description:

The description attribute is a character string that contains a description of an element. The content is at the discretion of the engine program supplier. It may be an empty string.

The description is output of listCategory, getDescription.
The description is not input to any API function.

5.7 Units:

The standard string for each type of unit is given in AS681. Categories will have an empty units string.

The supplier may make a parameter available to the calling program that allows it to change the Units family, e.g. between US customary and SI units, during the course of a run. If the supplier does make such a parameter available, and it is changed, the units string of each parameter needs to be obtained from the engine program. This can be done by calling either the listCategory function to relist the entire hierarchy, or calling the getDescription function to get the units string for each parameter individually.

The units string is output of listCategory, getDescription.
The units string is not input to any API function.

6. PARAMETER HIERARCHIES:

6.1 Overview:

Multiple hierarchies can be defined for a single engine program. A function is provided to list the hierarchies defined by the supplier for the engine program. The user can select which hierarchy to load when the engine program is initialized. Once initialized, the selected hierarchy remains in effect for the duration of the execution session.

Multiple keywords can be associated with a single hierarchy. It is recommended that suppliers associate one hierarchy with both the keyword "default" and an empty string keyword. This allows the user to initialize the program with the supplier's default hierarchy without having to know any of the specific hierarchy keywords defined for the engine program.

6.2 Required Hierarchies:

Engine programs that conform to this ARP must have at least one hierarchy that is associated with the keyword "AS681" This hierarchy contains only the root category. This category will have all parameters described in AS681, with the parameter names exactly as they are specified in AS681. The list of parameters in AS681 should be considered as the minimum that must appear in this hierarchy, even if the parameter is not used or calculated by the engine program. The hierarchy may contain additional parameters beyond those defined by AS681. Any additional parameter used in this hierarchy must appear in the root category. The only naming requirement for these additional parameters is that they conform to AS755.

7. TABULAR DATA:

The API setValue and getValue functions can only pass single data values or one-dimensional data arrays. Data tables of two or more dimensions can not be passed directly through the API functions. These data tables can be passed through the API by packing the data into a 1-D array. The data type attribute is used to distinguish regular 1-D array data from table data. Parameters that can contain data tables must have data type attributes of 21, 22, 23, or 24.

The index data necessary to unpack the 1-D data stream once it has passed through the API functions is placed in a separate integer array parameter, and is retrieved or set through a separate function call. These index array parameters always have a data type of 11, indicating they are an array of integers. A separate index array parameter is defined for each packed data stream, even if two or more packed data streams have identical index arrays.

7. (Continued):

More complex data structures, such as tabular functions, or graphs, can be represented by a group of parameters that define the content and structure of the multi-dimensional tables that make up the graph. For example, the commonly used graph for intake recovery can be represented by 5 parameters:

1. A 1-D stream for the values of mach number
2. A 1-D stream packed with the values of the multi-dimension referred flow table
3. A 1-D stream packed with the values of the multi-dimension ram recovery factor, eRam
4. A 1-D stream of integers containing the index information required to unpack the referred flow data stream
5. A 1-D stream of integers containing the index information required to unpack the eRam data stream.

It is recommended that a separate sub-category be defined to contain only the parameters necessary to define a table or graph. The name and description of the category become, in effect, the name and description of the table or graph. Each parameter will still have its own description, type and units attributes. With the subcategory as an organizing entity, further parameters could be included in the sub-category to define related information, such as interpolation and extrapolation methods.

Appendix B gives examples of how multi-dimensional tables and graphs can be packed into 1-D data streams for transmission through the setValue and getValue functions. The examples include the packing of uniformly and irregularly filled tables.

8. CHARACTER STRING REQUIREMENTES:

8.1 Accommodating Differences in Character String Format Between Programming Languages:

The character string formats used by different programming languages, and the different mechanisms used to pass character strings through the function call list are the principle reasons why a single set of API functions was not specified in this document. While it is possible for one function to pass character strings to or from a function written in another language, it is not simple.

This document recommends that the function in the calling program and the API function in the engine program are written in the same programming language. This will ensure that the character strings passed through the API functions are in a format that is native to both the calling and API function. If the language of the API functions, as agreed by the customer and supplier, is different from the core language of the engine program, it is the responsibility of the engine program to convert the character strings to the format native to the language of the API function.

The character string format and any character string requirements specific to each programming language are given in the appendix pertaining to that language.

8.2 Character String Lengths:

The following string lengths are recommended. Some programming languages, such as C, can be written such that additional memory is allocated if the character strings exceed the initial allocated length. Other languages, such as FORTRAN 77, use a fixed character length, which is set at compilation.

- Units = 32 characters
- Element name = 32 characters
- Parameter hierarchy keyword = 64 characters
- Parameter hierarchy keyword description = 512 characters
- Element path name = 256 characters
- Element description = 256 characters
- Element security classification = 64 characters
- NSI description = 256 characters
- Error and Warning description = 256 characters
- Engine program description = 4096 characters
- Engine program name = 64 characters
- Engine program version = 64 characters

The initial character string sizes were selected as multiples of 32. This was done so that the strings end on a word boundary for a 32-bit architecture, or half word boundary for a 64-bit architecture. The goal is to use core and disk memory efficiently. If the character string lengths need to be increased during a run, it is recommended that additional length be allocated in multiples of 32 characters.

9. FUNCTION ATTRIBUTES AND BEHAVIORS:

9.1 Function Names:

A unique set of function names will be defined for each programming language. This is done so that the user can determine the language type directly from the function name.

9.2 Element Path Names in Call List Arguments:

The full path name does not need to be used when calling the API functions. All API functions will accept relative path names. However, the concatenation of the default category path name plus the relative path name must be a valid full path name. If the path name begins with a backslash, it will be treated as a full path name and will not be appended to the default category path name.

9.3 Usage of Element Names and Direct Access IDs:

The direct access ID is used whenever an API function is called with a non-zero value for the ID. If the name is not an empty string, the ID is used to locate the element and the given name is validated against the name of element identified by the ID value. If the names do not match, the input name is used to locate the desired element and the ID corresponding to the named element is returned to the calling program. A warning flag is returned to notify the calling program that a match for the given ID was not found and that the ID corresponding to the given element name is being returned.

An empty name string is used to indicate that only an ID is being given. A valid name string corresponding to the given ID is returned. This is the only combination where precedence is given to the ID. What constitutes an empty string is dependent on the programming language being used. See the appropriate appendix for a definition.

An ID value of zero is used to indicate that only a name is being given, and that correspondence of the ID to the given name should not be checked. When the ID is zero, the ID that corresponds with the given name is returned through the call list. No error or warning message is generated.

Table 2 summarizes the action taken in each of the possible combinations of element name and ID.

TABLE 2 - Responses to Name and ID Combinations

	Valid ID	Invalid ID	ID=0
Valid Name	Use ID, verify that name matches	Use Name, return warning and corrected ID	Use Name, return corresponding ID
Invalid Name	Return error, stop function	Return error, stop function	Return error, stop function
Empty Name	Use ID, return corresponding name	Return error, stop function	Return error, stop function

9.3.1 API Function Return: Each API function will return a single integer value. It is used to indicate if errors and/or warnings occurred during the execution of the function. A zero return value indicates a successful execution of the function with no warnings or errors. A return value of one indicates that one or more warnings have been issued. A return value of two indicates that one or more errors have been issued. And a return value of three indicates that both warnings and errors have been issued.

9.3.2 Error and Warning Messages: Each API function call list includes two array arguments through which the function will return a numerical value for each error or warning and descriptive strings of those warnings or errors. A third argument gives the number of errors and warnings being returned by the function. The numerical value and descriptive string for each error or warning that is common to all programming languages is given in Section 10. Each programming language may have additional errors and warnings that are unique to that language. These additional errors and warnings are given with the function definitions in the appropriate appendix.

10. API FUNCTION DEFINITIONS:

10.1 Function Summary:

10.1.1 Structure of Function Definitions: The function names in this section are generic. Language specific function names are given in the appropriate appendix. The definition section for each API function has three parts: a description, an argument list, and a list of warning/error values and descriptions. The description outlines the expected behavior of the function. The argument list section describes the information passed and returned through the call list. The exact argument list and argument names may vary between programming languages. The common error and warning messages section lists the numerical value and description of those flags that are common to all programming languages. Error and warning flags that are specific to a programming language are specified in the appropriate appendix.

10.1.2 List of API Functions: Table 3 is a summary of the minimum list of API functions. The table lists those functions that may be implemented as a non-functional stub. A stub implementation provides a callable function but does not have a functional body. Instead, calls to a stub implementation immediately return with a warning message that the function has not been functionally implemented. A stub is needed so that all functions specified in the API are always present. The order presented represents a typical order of usage.

TABLE 3 - List of API Functions

Function Name	Description	Stub Acceptable
programInfo	Provides overall information about the engine program, including the program name and version. It also indicates whether the program supports element IDs, units, security classification, and descriptions, and whether it will flag parameters if a limit is reached/exceeded or if an operation violates the I/O status	No
listHierarchies	Lists the hierarchy keywords that are valid for the engine program	No
initializeProgram	Initializes the engine program with the selected element hierarchy. It must be called before all functions with the exception of programInfo and listHierarchies. It may be called only once per execution session	No
listCategory	Lists the contents of the specified category. The name, ID, type, I/O status, security classification, description and units are listed as appropriate and as available for each element	No
getDescription	Returns the element type, I/O status, security classification, description and units for an individual element	No
listChangedIOstatus	Lists the parameters that have had a change in I/O status since the previous time this function was called, or since initializeProgram was called	Yes
setDefaultCategory	Sets the default category	Yes
getDefaultCategory	Returns the name and ID of the current default category	Yes
setValue	A group of functions that sets the value of integer, single precision floating point, double precision floating point and character parameters. Different sets of functions are used for scalar and tabular parameters	No
runEngProgram	Runs the engine program with the set of parameter values as they exist in the engine program at the time of the call	No
getValue	A group of functions that gets the value of integer, single precision floating point, double precision floating point and character parameters. Different sets of functions are used for scalar and tabular parameters	No
listLimitParameters	List the parameters at or beyond a limit value, or that are not at the input value because of another parameter being held to a limit value	Yes
resetEngProgram	Resets all the input parameters to the baseline value defined by the supplier	Yes

10.2 programInfo - Overall Engine Program Information:

10.2.1 Description: The programInfo function provides overall information about the engine program. This includes strings representing the engine program name, engine program version or identifier number, and an engine program description. The engine program version argument may contain non-numeric characters.

The other arguments returned are integer flags. Each indicates the availability of information for the flagged attribute in the engine program. If a flag is true (an integer value of 1), all appropriate elements in the engine program have the information represented by the flag. If it is false (an integer value of 0), the flagged attribute is not supported by the engine program.

10.2.2 Call List Arguments:

TABLE 4 - Call List Arguments

Argument	Description	Type	Input/ Returned
programName	Engine program name string	string	Returned
programDescription	Engine program description string	string	Returned
programVersion	Engine program version label string	string	Returned
descriptionsReturned	Integer flag indicating if the engine will return non-empty string description strings for one or more parameters	int	Returned
unitsReturned	Integer flag indicating if the engine program returns the units strings for each dimensioned parameter	int	Returned
limitsFlagged	Integer flag indicating if the engine program will flag individual parameters if they are the limiting parameter, or if their input values not achieved because of a limit	int	Returned
IOchecked	Integer flag indicating if the engine program checks the I/O status when setting or retrieving a parameter value and returns a warning if the I/O status is not either Input/Active when setting or Calculated/Active when retrieving a parameter value	int	Returned
IDsupported	Integer flag indicating if the engine program has a non-zero ID for all elements	int	Returned
NSIdescriptions	Integer flag indicating if the engine program will return descriptive strings of any NSI that occurs	int	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.2.3 Common Error and Warning Messages: No common error or warning messages.

10.3 listHierarchies - List Available Hierarchies:

10.3.1 Description: The listHierarchies function returns an array of strings containing the keywords available for use in the initializeProgram function. It also returns an array of strings containing a description of the hierarchy associated with each keyword. More than one keyword may be associated with a given hierarchy. Thus, there may be more keywords than actual hierarchies in the engine program. The maximum number of keyword, unless otherwise agreed to, is 25.

The list of keywords must contain a keyword that is an empty string. This is the hierarchy selected if an empty string is passed to the initializeProgram function. The list should also include the keyword "default". Both empty string and default keywords should be associated with the same hierarchy. The list must also contain the keyword "AS681". This keyword is associated with the AS681 hierarchy described in section 6.

10.3.2 Call List Arguments:

TABLE 5 - Call List Arguments

Argument	Description	Type	Input/ Returned
numKeys	The number of keywords. Maximum of 25	int	Returned
keywords	Array of character strings containing the keywords associated with the parameter hierarchies available in the engine program	string array	Returned
keywordDescriptions	Array of character strings containing the descriptions of hierarchies associated with the keywords	string array	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.3.3 Common Error and Warning Messages: No common error or warning messages

10.4 initializeProgram - Initialize Engine Program:

10.4.1 Description: The initializeProgram function performs any initialization required by the engine program. This function must be called before any other API function in the engine program, with the exception of the programInfo and listHierarchies functions. The initializeProgram function may be called only once per execution session. An error will be returned if this function is called more than once. All functions, other than programInfo and listHierarchies will return an error if they are called before this function, and therefore the engine program is responsible for retaining the fact that initializeProgram has been called.

The string given for the "keyword" argument must be from the list of hierarchy keywords defined by the supplier. An empty string keyword may always be passed to this function, since it must be defined for all engine programs. An empty keyword will result in the supplier's default hierarchy being used. Otherwise, a non-empty string, valid keyword must be supplied to initialize the engine program.

After initializeProgram returns, the selected hierarchy has been loaded, all the input parameters have been set to a baseline value defined by the supplier, and the default category has been set to be the root category. The engine program is then ready to accept user input. The baseline values of the input parameters should represent a valid set of values, and the engine program should be able to execute without additional user input. The calculated parameters are not guaranteed to contain valid values that correspond to the input parameter baseline values until the runEngProgram function has been successfully executed.

10.4.2 Call List Arguments:

TABLE 6 - Call List Arguments

Argument	Description	Type	Input/ Returned
keyword	Character string containing the name of the hierarchy to be loaded	string	Input
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.4.3 Common Error and Warning Messages:

TABLE 7 - Common Error and Warning Messages

Error/Warning Number	Description
101	ERROR: Unknown Keyword
102	ERROR: Program already initialized
103	ERROR: Initialization unsuccessful

10.5 listCategory - List Category Contents:

10.5.1 Description: The listCategory function lists the contents of a single category. It returns the number of elements listed and arrays containing the name, ID and all other attributes of each element in the category. The calling program passes in the name and/or category ID of the category to be listed.

On the first call to list each category, the calling program sets nStart equal to one and passes in the maximum number of element names it can accept in the returned arrays through the maxElements argument. If the number of names to be listed exceeds the maximum value specified by the calling program, then the first maxElements number of elements is returned, the value of the nStart argument is set to the index of the next element to be listed, and a message is returned to indicate that additional elements remain to be listed.

When this occurs, the calling program is expected to make another call to listCategory, again indicating the maximum number of elements that can be accepted using the maxElements argument and pass back in the value of nStart returned by the function on the previous call. The remaining elements or next maxElements number of elements are returned. This cycle will continue until all the elements in that category have been listed. On the call to listCategory that lists the last elements in the category the value of nStart is reset to a value of one.

10.5.2 Call List Arguments:

TABLE 8 - Call List Arguments

Argument	Description	Type	Input/ Returned
categoryName	Name of category to list	string	Input/ Returned if input empty string and ID not equal zero
categoryID	Category Identifier	int	Input/ Returned if input equals zero
maxElements	Maximum number of elements that the function can return as specified by the calling program	int	Input
nStart	Index number at which listing is to begin	int	Input/ Returned
numElements	Number of elements returned	int	Returned
elementNames	Array of element names in the category	string array	Returned
elementIDs	Array of direct access identifiers for elements in the category	int array	Returned
elementTypes	Array of element type flags flag value: 0 Category 1 Integer 2 Single Precision Float 3 Double Precision Float 4 Character String +10 Array data +20 Packed table data	int array	Returned
elementIOflags	Array of element I/O status flags flag value: 0 Undefined (category) 1 Input/Active 2 Calculated/Active 3 Input/Inactive 4 Calculated/Inactive	int array	Returned
elementClassifications	Array of classification strings	string array	Returned
elementDescriptions	Array of element descriptions strings	string array	Returned
elementUnits	Array of element unit strings	string array	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.5.3 Common Error and Warning Messages:

TABLE 9 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Given categoryName and categoryID do not agree
2	WARNING: Number of elements to be listed is greater than maxElements
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Invalid categoryName
103	ERROR: Invalid categoryID
104	ERROR: Value of nStart is out of range

10.6 getDescription - Get Element Attributes:

10.6.1 Description: The getDescription function returns all information available for a specific element. The attributes returned by this function are the element type integer, I/O status integer, classification level, description string, and units string. The unit strings returned are as specified in AS681.

10.6.2 Call List Arguments:

TABLE 10 - Call List Arguments

Argument	Description	Type	Input/ Returned
elementName	Element Name	string	Input/ returned if input empty string and ID not equal zero
elementID	Element Identifier	int	Input/ returned if input equals zero
elementType	Element type flag flag value: 0 Category 1 Integer 2 Single Precision Float 3 Double Precision Float 4 Character String +10 Array data +20 Packed table data	int	Returned
elementIOflag	Element status flag flag value: 0 Undefined (category) 1 Input/Active 2 Calculated/Active 3 Input/Inactive 4 Calculated/Inactive	int	Returned
elementClassification	Classification string	string	Returned
elementDescription	Element description string	string	Returned
elementUnits	Element unit string	string	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.6.3 Common Error and Warning Messages:

TABLE 11 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Given elementName and elementID do not agree
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Invalid elementName
103	ERROR: Invalid elementID

10.7 listChangedIOstatus - Get Parameters with Changed I/O Status:

10.7.1 Description: The listChangedIOstatus function returns the full path name and I/O status of any parameter that has changed I/O status since the last time this function was called (or since initializeProgram was called for the first call to this function). Changing the values of some parameters may cause the I/O status of other parameters in the engine program to change. All set functions contain a value of the "flag return" flag that indicates that the I/O status of one or more parameters has changed as a result of the function call.

The engine program will accumulate a list of those parameters that changed I/O status and the new I/O status value. If the I/O of a parameter has changed more than once since the last call to this function, only the last I/O status value is returned.

This function should be included in all engine programs, but it does not have to be fully implemented. Coverage of I/O status changes does not have to be exhaustive. listChangedIOstatus can be written to report only the changes in I/O status that can be known prior to program execution. This is similar to the I/O status information given in a customer's manual. An example of this situation is the variable that determines how the flight condition is specified. Changing the value of this variable changes the I/O status of a number of other variables in a known way.

On the first call to list each category, the calling program sets nStart equal to one and passes in the maximum number of parameter names it can accept in the returned arrays through the maxParameters argument. If the number of names to be listed exceeds the maximum value specified by the calling program, then the first maxParameters number of elements is returned, the value of the nStart argument is set to the index of the next element to be listed, and a message is returned to indicate that additional parameters remain to be listed.

When this occurs, the calling program is expected to make another call to listChangedIOstatus, again indicating the maximum number of elements that can be accepted using the maxParameters argument and pass back in the value of nStart returned by the function on the previous call. The remaining parameters or next maxParameters number of parameters are returned. This cycle will continue until all the affected parameters have been listed. On the call to listChangedIOstatus that lists the last elements in the category the value of nStart is reset to a value of one.

10.7.2 Call List Arguments:

TABLE 12 - Call List Arguments

Argument	Description	Type	Input/ Returned
maxParameters	Maximum number of parameters that the function can return as specified by the calling program	int	Input
nStart	Index number at which listing is to begin	int	Input/ Returned
numberChanged	Number of parameters with changed I/O status listed in the returned arrays	int	Returned
parameterNames	Array of full parameter path names	string array	Returned
parameterIDs	Array of parameter identifiers	int array	Returned
parameterIOflags	Array of Parameter I/O status flags flag value: 1 Input/Active 2 Calculated/Active 3 Input/Inactive 4 Calculated/Inactive	int array	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.7.3 Common Error and Warning Messages:

TABLE 13 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Function not implemented
2	WARNING: Number of parameters to be listed is greater than maxParameters
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Value of nStart is out of range

10.8 setDefaultCategory - Set Default Category:

10.8.1 Description: The setDefaultCategory function sets the default category. When the engine program receives a relative path name, name matching starts at this internally stored default category. A full path name must be used when specifying a new default category. The path name of the root category is a string containing only a backslash character. If the categoryName is an empty string and the categoryID corresponds to a valid category, then the name of the corresponding category is returned through the categoryName argument.

10.8.2 Call List Arguments:

TABLE 14 - Call List Arguments

Argument	Description	Type	Input/ Returned
categoryName	Category name to make the default category	string	Input/ returned if input empty string and ID not equal zero
categoryID	Category Identifier to make the default category	int	Input/ returned if input equals zero
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.8.3 Common Error and Warning Messages:

TABLE 15 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Function not implemented
2	WARNING: Given categoryName and categoryID do not agree
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Invalid categoryName
103	ERROR: Invalid categoryID
104	ERROR: categoryName is not an absolute path name

10.9 getDefaultCategory - Get Default Category:

10.9.1 Description: The getDefaultCategory function returns the full path name and category ID of the current default category.

10.9.2 Call List Arguments:

TABLE 16 - Call List Arguments

Argument	Description	Type	Input/ Returned
categoryName	Default category name	string	Returned
categoryID	Default category identifier	int	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.9.3 Common Error and Warning Messages:

TABLE 17 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Function not implemented
101	ERROR: initializeProgram not called. Program not initialized

10.10 setValue - Set Parameter Values:

10.10.1 Description: The setValue functions will set the parameter to the specified value. A separate function is required to set integer, single and double precision floating point, and character data as well as to set arrays of those types. See the appropriate appendix for implementation details.

Multidimensional data arrays can not be directly set through this API. Instead these functions can only accept 1-D arrays. However, multidimensional data can be packed into a 1-D array. A separate parameter is used to pass the index array necessary to unpack the 1-D data array. The arrays for both parameters must be passed to the engine program in order for it to unpack the multidimensional data. See Section 7 for details on how to pack and unpack multidimensional data into and out of a 1-D array.

10.10.1 (Continued):

The setValue functions will set the value in the engine program even if the I/O status is not input/active. While setting the value of an inactive input or a calculated parameter will not change the calculation results, it may mislead users. They may not notice that the value they set is being ignored. The engine program should, therefore, return a warning flag if the parameter being set has an I/O status of other than input/active.

If changing the value of the parameter causes the I/O status of any parameter in the engine program to change, a flag is returned through the IOchanged argument. It is the user's responsibility to query the listChangedIOstatus function to determine which parameters have changed and to find out their new I/O status value.

10.10.2 Call List Arguments:

TABLE 18 - Call List Arguments

Argument	Description	Type	Input/ Returned
parameterName	Parameter Name	string	Input/ returned if input empty string and ID not equal 0
parameterID	Parameter Identifier	int	Input/ returned if input equals zero
value	Value or array being set	int single double string int array single array double array string array	Input
valSize	Size of the array of values being passed. Only applies to setValue functions passing an array of values	int	Input
IOchanged	Indicator that setting the parameter caused one or more parameters to change I/O status zero = FALSE, non-zero = TRUE	int	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.10.3 Common Error and Warning Messages:

TABLE 19 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Given parameterName and parameterID do not agree
2	WARNING: Inactive input parameter set
3	WARNING: Calculated parameter set
4	WARNING: Value exceeded limit, value used
5	WARNING: Value exceeded limit, value set to limit
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Invalid parameterName
103	ERROR: Invalid parameterID
104	ERROR: Number or size of table dimensions incompatible with engine program table being set

10.11 runEngProgram - Engine Program Execution:

10.11.1 Description: The runEngProgram runs the current case, as defined by the values of the input parameters when this function is called, and returns any numerical status indicators in the NSI array argument. The values of the NSI will conform to AS681. A description string for each NSI will be passed back as well. The maxNSI argument sets the upper limit on the number of NSI values that the calling program is able to accept through the NSI and NSIdesc arguments. If the number of NSIs generated exceeds the specified NSI return array size, then the method specified in AS681 for determining which NSI to return is used. The NSI description strings will follow the same pattern.

The limitsFlagged argument indicates if any parameters exceeded or were held to a limit value. If this argument is true, then the listLimitParameters function can be called to return a list of these parameters. This list will include any parameter where the value used by the engine program was different than the input value as a result of other parameters reaching or exceeding their limit values.

10.11.2 Call List Arguments:

TABLE 20 - Call List Arguments

Argument	Description	Type	Input/ Returned
maxNSI	The maximum number of NSIs per case that runEngProgram is allowed to return	int	Input
numNSI	Number of NSI values returned	int	Returned
NSI	Array of NSI values	int array	Returned
NSIdesc	Array of descriptive strings for each of the numerical NSI values	string array	Returned
limitsFlagged	A flag indicating if any parameter is at or beyond a limit or modified due to a limit. Zero = TRUE, non-zero = FALSE	int	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.11.3 Common Error and Warning Messages:

TABLE 21 - Common Error and Warningn Messages

Error/Warning Number	Description
1	WARNING: One or more non-fatal NSI flags returned
2	WARNING: Number of NSIs exceeded return array size
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: One or more fatal NSI flags returned

10.12 getValue - Get Parameter Values:

10.12.1 Description: The getValue functions return the value for the named parameter. A separate function is required to get integer, single and double precision floating point, and character data as well as to get arrays of those types. Different programming languages accommodate this requirement in different ways. See the appropriate appendix for implementation details.

For the getValue functions that return an array of values, the initial call to this program by the calling program passes in nStart equal one and the maximum array sizes it can accept. If the number of values in the array exceeds the maximum value specified by the calling program, then the first maxvalues number of values are returned, the value of the nStart argument is set to the index of the next value to be listed, and a message is returned through the function return value to indicate that additional values remain in the array.

When this occurs, the calling program is expected to make another call to getValue, again indicating the maximum number of elements that can be accepted using the maxValues argument. The remaining parameters or next maxValues number of parameters are returned. This cycle will continue until all the values in the array have been returned. On the call to getValue that lists the last elements in the category the value of nStart is reset to a value of one.

Multidimensional data arrays can not be directly returned through this API. Instead these functions can only accept 1-D arrays. However, multidimensional data can be packed into a 1-D array. A separate parameter is used to return the index array necessary to unpack the 1-D data array. Both parameters must be retrieved from the engine program in order to unpack the multidimensional data. See Section 7 for details on how to pack and unpack multidimensional data into and out of a 1-D array.

10.12.2 Call List Arguments:

TABLE 22 - Call List Arguments

Argument	Description	Type	Input/ Returned
parameterName	Parameter Name	string	Input/ returned if input empty string and ID not equal 0
parameterID	Parameter Identifier	int	Input/ returned if input equals zero
maxValues	Maximum number of values that the calling program can accept through the getValue functions that return an array	int	Input
nStart	Index number at which listing is to begin. Applies to getValue functions that return an array	int	Input/ Returned
value	Value(s) being returned	int single double string int array single array double array string array	Returned
valSize	Size of the array of values being passed. Only applies to getValue functions passing an array of values	int	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.12.3 Common Error and Warning Messages:

TABLE 23 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Given parameterName and parameterID do not agree
2	WARNING: number of values exceeds maxValues
3	WARNING: IOstatus of requested parameter is input/inactive
4	WARNING: IOstatus of requested parameter is calculated/inactive
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Invalid parameterName
103	ERROR: Invalid parameterID
104	ERROR: Value of nStart is out of range

10.13 listLimitParameters - List Parameters At or Beyond Specified Limits:

10.13.1 Description: The listLimitParameters function returns a list of parameters that are either at or beyond their limit value or that are not at their input value because another parameter is being held to a limit value. An example might be an input value of fuel flow that causes the N2 speed to exceed its limit value. This function would return the name of the N2 parameter and a limit flag value of 1 to indicate it is above its maximum value. If the fuel flow were such that N1 and exhaust gas temperature were above their limits as well, then all three parameters would be listed in the return of this function.

If, however, the fuel flow was cut back from the set value in order to hold the N2 at its limit, then this function would return the N2 and fuel flow parameter names. The limit flag value for N2 would be 2 to indicate that it is being held at the maximum value. The limit flag value for fuel flow would be 5 to indicate that it is at a value below its input value.

On the first call to this function, the calling program passes in nStart equal one and the maximum number of parameter names it can accept in the returned arrays through the maxParameters argument. If the number of names to be listed exceeds the maximum value specified by the calling program, then the first maxParameters number of elements is returned, the value of the nStart argument is set to the index of the next element to be listed, and a message is returned to indicate that additional elements remain to be listed.

When this occurs, the calling program is expected to make another call to listLimitParameters, again indicating the maximum number of elements that can be accepted using the maxParameters argument and pass back in the value of nStart returned by the function on the previous call. The remaining parameters or next maxParameters number of parameters are returned. This cycle will continue until all the elements in that category have been listed. On the call to listCategory that lists the last elements in the category the value of nStart is reset to a value of one.

10.13.2 Call List Arguments:

TABLE 24 - Call List Arguments

Argument	Description	Type	Input/ Returned
maxParameters	Maximum number of parameters that the function can return as specified by the calling program	int	Input
nStart	Index number at which listing is to begin	int	Input/ Returned
numParameters	Number of parameters for which limit information is listed in the returned arrays.	int	Returned
parameterNames	Array of path names of parameters at or beyond a limit or affected by another parameter reaching a limit	string array	Returned
parameterIDs	Array of corresponding parameter Identifiers	int array	Returned
limitFlags	Array of flags indicating how each parameter value was affected 1 Parameter value above maximum 2 Parameter value held at maximum 3 Parameter value below minimum 4 Parameter value held at minimum 5 Parameter value decreased compared to the input value 6 Parameter value increased compared to the input value	int array	Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.13.3 Common Error and Warning Messages:

TABLE 25 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Function not implemented
2	WARNING: Number of parameters to be listed is greater than maxParameters
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Value of nStart is out of range

10.14 resetEngProgram - Reset Engine Program Values:

10.14.1 Description: The resetEngProgram function resets all input parameters to the baseline value defined by the supplier. The default category remains the same as when this function was called. It is not set back to the root category. The calculated parameters are not guaranteed to contain valid values that correspond to the input parameter values until the engine program has been executed though the runEngProgram function.

10.14.2 Call List Arguments:

TABLE 26 - Call List Arguments

Argument	Description	Type	Input/ Returned
numErrors	Number of errors and warnings issued by the function	int	Returned
errorValues	Array containing the numerical values of the errors or warnings issued by the function.	int array	Returned
errorMessages	Array of descriptive strings of any errors or warnings issued by the function	string array	Returned

10.14.3 Common Error and Warning Messages:

TABLE 27 - Common Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Function not implemented
101	ERROR: initializeProgram not called. Program not initialized
102	ERROR: Reset unsuccessful

APPENDIX A
EXAMPLE OF VARIABLE HIERARCHIES

The following are sample variable hierarchies that are supported by this API. Hierarchies are not limited to these structures. The organization of the hierarchy is at the discretion of the engine program supplier.

A.1 Example 1: AS681 Hierarchy:

This first example is a partial listing of the required hierarchy with the keyword of "AS681". It is a hierarchy with only the root category. It contains all the variables specified as the minimum variable set by AS681. The variable names are as given in AS681. The names of variables beyond the required minimum specified in AS681 must follow the name construction rules specified in AS755. The default category must be the root category since it is the only category in this hierarchy. As such the parameter names can be referred to without a leading backslash.

\CASE
\TITLE
\ZALT
\ZDTAMB
\ZDT1A
\ZERM1A
\ZPWXH
\ZP1A
\ZRC
[…]
\AE18
\ALT
\CLASS
\DTAMB
\ERAM1
\FG
\FGI
\FHV
\FN
\NSI
\P7
\PB3
\WB3Q

A.2 Example 2: Structured Hierarchy:

This second example shows how engine program parameters could be organized in a structured hierarchy based around the components in an engine. It shows parameters that would not normally be visible in a customer engine program as a way to illustrate several levels of nested subcategories in a structured hierarchy. Some parameters, such as Fnet, are contained directly within the root category, while others like \Core\HPC\Map\ReCorrection\etaReScalar, are in several layers of nested categories.

```
\Fnet
\TSFC
\OPR
\Ambient\       alt
                MN
                dTamb
                Pamb
                Tamb
\Inlet\         Fram
                eRam
                eRamBase
                FluidIn\        Pt
                                Tt
                                W
\Core\          HPC\            PR
                                TR
                                eta
                                Nref
                                Map\            etaDesign
                                                etaMap
                                                etaAdder
                                                etaScalar
                                                NrefDesign
                                                NrefMap
                                                ReCorrection\   ReDesign
                                                                etaReScalar
                Burner\         FAR
                                Wf
                                FluidOut\       Tt
                HPT\            PR
                                TR
                                Map\            PRdesign
                                                PRmap
                                                PRadder
                                                PRscalar
                                eta
\Nozzle\        Fg
                Cv
                Cfg
                AR
                Aout
                Ath
\CustomerBleeds\                High\           W
                                                Tt
                                                Pt
                                Low\            W
                                                Tt
                                                Pt
\AccessoryShaft\                Pw
                                Trq
                                N
```

APPENDIX B
EXAMPLE OF TABULAR DATA TO 1-D DATA ARRAY CONVERSION

This Appendix gives examples of how 2-D and higher order data tables and graphs can be packed into 1-D data arrays that can be sent through the API setValue and getValue functions. Different methods are used to pack uniformly filled and irregularly filled data tables into the 1-D data stream. Examples of both are given. Also given are examples of how complete function graphs, such as those for ram recovery, can be packed into a set of arrays that can be individually passed through the API functions.

CREATING THE 1-D DATA STREAM

The data from the table is simply written to the 1-D array in row order.

CREATING THE INDEX ARRAY

In addition to the 1-D data stream, a 1-D index array for each table is required. The index array contains all the information necessary to unpack the 1-D data stream and restore the data to its original structure.

The naming convention for the index arrays is to simply affix "index" to the name of the 1-D array containing the packed data.

The first three elements of the index array are always the same.

- Element 1: Dimensionality of the table.
- Element 2: Flag indicating if the table is uniformly (a zero) or irregularly filled (a one).
- Element 3: Number of subtables of dimensionality n-1.

After the third element, the elements in the index array differ between uniformly and irregularly filled tables. Uniformly filled tables have the same number of points in each line, the same number of lines in each 2-D table, the same number of 2-D subtables in each 3-D table and so forth. As a result, the index for a uniformly filled table needs only to contain this single piece of information for each level in the table.

Irregularly filled tables, on the other hand, do not have this inherent predictability. The index array for an irregular table must contain information giving the number of data points in each line and the number of lines in each subtable. For example, one line might contain 5 points while the next might contain 8. There might be 10 lines in one 2-D subtable and 7 in the next.

Since, in practice many data tables may be irregularly filled at the highest dimensions, but uniformly filled at lower dimensions, the packing algorithm illustrated in the following examples allows for the index array to also be a mix. Thus, at each level in the table there will be a flag to indicate if that level is uniformly or irregularly filled. While an irregularly filled table may have uniformly filled subtables, the converse is not allowed. Once a table or subtable is declared to be uniformly filled, that entire table or subtable must have that structure.

Please note that sections 1 through 4 are simply examples of how 2-D and 3-D tables of numbers can be packed into a 1-D array. The tables in these section have no physical meaning. They are simply a series of number organized into a 2-D or 3-D table. Section 5 and 6 illustrate how this methodology applies to packing actual table data using a map of eRam as an example.

B.1 Uniformly Filled 2-D Table:

The following is an example of converting a uniformly filled table from tabular data to a 1-D data stream and an index array.

Assume that we have the following 2-D table:

```
        1   2   3   4   5
A =    11  12  13  14  15
      101 102 103 104 105
```

Note that all elements in all rows are filled. The data from the table is packed into the 1-D array in row order. The 1-D data array would then be as follows:

```
B = [1, 2, 3, 4, 5, 11, 12, 13, 14, 15, 101, 102, 103, 104, 105]
```

The resulting index array for the 2-D uniformly filled table is:

```
Bindex = [2, 0, 3, 5]
```

The following is the meaning of each of the 4 elements in Bindex.

Element 1 = 2 A is a 2-D table.
Element 2 = 0 A is a uniformly-filled table.
Element 3 = 3 There are three rows in A.
Element 4 = 5 There are 5 data elements in each row.

B.2 Uniformly Filled 3-D Table:

The following is the index array for a uniformly filled 3-D table that has 4 subtables of the same structure as the 2-D table A in the example above.

```
Bindex = [3, 0, 4, 3, 5]
```

B.3 Irregularly Filled 2-D Table:

The following is an example of converting an irregularly-filled table into a 1-D data stream and the resulting index array. Since the structure of the data can vary at each level in the table, considerably more index information is required for an irregular data table.

Assume that we have the following 3-D table, Z which consists of sub-tables X and Y:

```
        1   2   3   4            101 102 103 104 105 106
Z =     5   6   7                107 108 109 110
        8   9
                X                        Y
```

The 1-D data array written in row order would be as follows:

```
C = [1, 2, 3, 4, 5, 6, 7, 8, 9, 101, 102,
     103, 104, 105, 106, 107, 108, 109, 110]
```

The index array Cindex for table Z is as follows:

```
Cindex = [3, 1, 2,    1, 3, 4, 3, 2,    1, 2, 6, 4]
```

Element 1 = 3 Z is a 3-D table.
Element 2 = 1 Z is an irregularly filled table.
Element 3 = 2 There are two 2-D sub-tables in Z.
Element 4 = 1 The X sub-table is irregular.
Element 5 = 3 The X sub-table has three rows.
Element 6 = 4 The first row in X has 4 elements.
Element 7 = 3 The second row in X has 3 elements.
Element 8 = 2 The third row in X has 2 elements.
Element 9 = 1 The Y sub-table is irregular.
Element 10 = 1 The Y sub-table has 2 rows.
Element 11 = 6 The first row in Y has 6 elements.
Element 12 = 4 The second row in Y has 4 elements.

Note that elements 4 to 8, which define the first sub-table, and elements 9 to 12, which define the second sub-table, are similar to an index array for an irregular 2-D table, except that the dimension of the table is known and therefore not repeated. This enables complex tables to be built up from lower order tables. Also it enables the sub-tables to be either regular or irregular.

B.4 Irregularly Filled 3-D Table with Uniformly and Irregularly Filled 2-D Subtables:

This example shows the index array for an irregular 3-D table, which contains both regular and irregular sub-tables. There are four sub-tables, the first, second, and fourth are regular, and the third is irregular.

The first sub-table contains 9 rows and there are 6 elements per row.

The second sub-table contains 8 rows and there are 7 elements per row.

The third sub-table is the same as subtable Y in EXAMPLE 2 above.

The fourth sub-table contains 5 rows and there are 12 elements per row.

The index for the complete table is:

```
Dindex = [3, 1, 4,   0, 9, 6,   0, 8, 7,
          1, 2, 6, 4,   0, 5, 12]
```

B.5 eRam Function with Uniformly Filled Tables:

This is an example of a complete tabular function represented by a group of parameters. The function is the typical inlet recovery table of ERAM = f (W1R, XM). Each row in ERAM represents the data for one Mach number. The values in each row are the value of ERAM for each of the Flow function. Since the ERAM for each Mach number correspond to the same values of Flow function, the result is a uniformly filled ERAM table.

Parameter 1 is a 1-D array of values of Mach number:

```
XM = [0, .3, .8]
```

Parameter 2 is a 1-D array of values of Flow function:

```
W1R = [100, 120, 140, 170, 195]
```

Parameter 3 is a 2-D table of values of Recovery Factor with 3 rows of 5 values each:

```
ERAM =  [.988, .981, .970, .950, .933,
         .990, .985, .978, .967, .950,
         .990, .985, .980, .972, .960]
```

B.5 (Continued):

Parameter 4 is a 1-D index to unpack the data in ERAM:

```
ERAMindex = [2, 0, 3, 5]
```

indicating that this is a table of 2 dimensions (index 1=2) that is uniformly filled (index 2=0) with 3 rows (index 3=3) with 5 data points each (index 4=5).

B.6 eRam Function with Irregularly Filled Tables:

This example shows an irregular ERAM function, where the number and value of Flow function and the corresponding values of ERAM are different for each Mach number. The function is as follows:

For XM= 0:

```
W1R = 100, 120, 140, 170, 195,

ERAM = .988, .981, .970, .950, .933,
```

For XM =.5:

```
W1R = 105, 125, 145, 175,

ERAM = .989, .984, .977, .966,
```

For XM =.85:

```
W1R = 110, 130, 150,

ERAM = .986, .981, .974,
```

This function can be represented by 1 1-D table and 2 irregularly filled 2-D tables.

```
XM    0.0   0.5   0.85

      100   120   140   170   195
W1R   105   125   145   175
      110   130   150

       .988  .981  .970  .950  .933
ERAM   .989  .984  .977  .966
       .986  .981  .974
```

To store this irregular function we need 5 1-D parameters. The XM parameter is simply the three Mach numbers, for which an index parameter is not needed. The data type of this parameter is 12 (array of single precision floating point numbers).

B.6 (Continued):

The W1R and ERAM parameters are the result of packing the Flow Function values into one array and the Recovery Factor into another. The data type for these arrays is 22 (table of single precision floating point numbers). The W1R and ERAM parameters require index information in order to unpack the data.

Even though the index data for W1R and ERAM are identical, separate index arrays are specified. This is done to avoid making exceptions to the general algorithm rule of one packed array to one index array. The data type of the index arrays is 12.

```
XM = [0, .5, .85]

W1R = [100, 120, 140, 170, 195, 105,
       125, 145, 175, 110, 130, 150]

W1Rindex = [2, 1, 3, 5, 4, 3]

ERAM = [.988, .981, .970, .950, .933, .989,
        .984, .977, .966, 986, .981, .974]

ERAMindex = [2, 1, 3, 5, 4, 3]
```

The meaning of the values in both the W1Rindex and ERAMindex arrays are:

Element 1 = 2 This is a 2-D table.
Element 2 = 1 This is an irregularly-filled table.
Element 3 = 3 There are 3 1-D sub-tables.
Element 4 = 5 The first sub-table has 5 elements.
Element 5 = 4 The second sub-table has 4 elements.
Element 6 = 3 The third sub-table has 3 elements.

APPENDIX C
FORTRAN 77 IMPLEMENTATION

C.1 Overview:

This appendix gives only the additional information required for FORTRAN 77 implementations of this API. The FORTRAN 77 implementation differs from the implementation in other programming languages in two key aspects. First is that the string lengths and array sizes in the FORTRAN 77 API functions are coded into the functions. This means that the maximum size for all strings must be known in advance. Second is the need to have short function and argument names. The strict ANSI FORTRAN 77 specification limits function names to 6 characters. However, creating intelligible 6 character function and argument names is very difficult. Few compilers adhere to this limit. The majority allow at least 7 character names. This API takes advantage of this common deviation and defines functions and argument names up to 7 characters.

FORTRAN 77 is unable to use differences in the number and type of the call list arguments to discriminate between functions with the same name. Therefore each set and get function for each parameter type has a different name. The same set of array passing functions is used for both array and packed tabular data, since these types differ only in the structure of the contents of the array.

C.2 Character String Format:

All character string lengths must be defined in the code. If the actual string length is less than the allocated length, then ASCII blank characters are appended to the string to bring it to the allocated length. An empty string is specified by a string containing all blank characters.

C.3 Character String Lengths:

The recommended character string lengths for a FORTRAN implementation are the same as given in section 8.2

C.4 API Function Definitions in FORTRAN 77:

The following gives the call list and argument declaration for the API functions as they would appear in a FORTRAN function. FORTRAN functions are used rather than subroutines because they return a value and because functions are the normal organizational unit in most other languages. The call list argument names are contractions of the argument names given in the general API function definition. They are, therefore, not repeated here. All common error and warning messages may be returned by the FORTRAN function. Any error or warning messages that are unique to a FORTRAN 77 implementation are noted here.

C.4.1 PGMINFO - Overall Engine Program Information:

C.4.1.1 Description: The general description of this function is applicable to the FORTRAN 77 implementation.

C.4.1.2 Function Definition:

```
INT FUNCTION PGMINFO (MNAME, MDESC, VERSION, DESC, UNITS, LIMITS,
          IOCHECK, IDSUP, NSIDESC, NUMERR, ERRVAL, ERRMSGS)

CHARACTER*(*) MNAME, MDESC, VERSION, ERRMSGS(*)
INTEGER DESC, LIMITS, IOCHECK, UNITS, IDSUP, NSIDESC, NUMERR,
          ERRVAL(*)
```

C.4.1.3 FORTRAN 77-Specific Error and Warning Messages:

TABLE C1 - FORTRAN 77-Specific Error and Warning Messages

Error/Warning Number	Description
2	WARNING: Insufficient space allocated for returning strings, strings truncated

C.4.2 LISTHIR - List Available Hierarchies:

C.4.2.1 Description: The general description of this function is applicable to the FORTRAN 77 implementation.

C.4.2.2 Function Definition:

```
INT FUNCTION LISTHIR(NUMKEYS, KEYWORD, KEYDESC, NUMERR, ERRVAL,
          ERRMSGS)

CHARACTER*(*) KEYWORD(*), KEYDESC(*), ERRMSGS(*)
INTEGER NUMKEYS, NUMERR, ERRVAL(*)
```

C.4.2.3 FORTRAN 77-Specific Error and Warning Messages:

TABLE C2 - FORTRAN 77-Specific Error and Warning Messages

Error/Warning Number	Description
1	WARNING: Insufficient space allocated for returning strings, strings truncated

C.4.3 INITPGM - Initialize Engine Program:

C.4.3.1 Description: The general description of this function is applicable to the FORTRAN 77 implementation.

C.4.3.2 Function Definition:

```
INT FUNCTION INITPGM(KEYWORD, NUMERR, ERRVAL, ERRMSGS)

CHARACTER*(*) KEYWORD, ERRMSGS(*)
INTEGER NUMERR, ERRVAL(*)
```

C.4.3.3 FORTRAN 77-Specific Error and Warning Messages: No FORTRAN specific warning or error messages.

C.4.4 LISTCAT - List Category Contents:

C.4.4.1 Description: The general description of this function is applicable to the FORTRAN 77 implementation.

C.4.4.2 Function Definition:

```
INT FUNCTION LISTCAT (CNAME, CID, MAXELES, NSTART, NUMELES,
           NAME, ID, TYPE, IOSTAT, CLASS, DESC, UNITS, NUMERR,
           ERRVAL, ERRMSGS)

INTEGER     CID, MAXELES, NSTART, NUMELES, ID(MAXELES),
           TYPE(MAXELES), IOSTAT(MAXELES), CLASS(MAXELES),
           NUMERR, ERRVAL(*)
CHARACTER*(*) CNAME, NAME(*), DESC(*), UNITS(*) , ERRMSGS(*)
```

C.4.4.3 FORTRAN 77-Specific Error and Warning Messages:

TABLE C3 - FORTRAN 77-Specific Error and Warning Messages

Error/Warning Number	Description
3	WARNING: Insufficient space allocated for returning strings, strings truncated

C.4.5 GETDESC - Get Element Attributes:

C.4.5.1 Description: The general description of this function is applicable to the FORTRAN 77 implementation.

C.4.5.2 Function Definition:

```
INT FUNCTION GETDESC (NAME, ID, TYPE, IOSTAT, CLASS, DESC, UNITS,
          NUMERR, ERRVAL, ERRMSGS)

INTEGER ID, TYPE, IOSTAT, NUMERR, ERRVAL(*)
CHARACTER*(*) NAME, DESC, UNITS, CLASS, ERRMSGS(*)
```

C.4.5.3 FORTRAN 77-Specific Error and Warning Messages:

TABLE C4 - FORTRAN 77-Specific Error and Warning Messages

Error/Warning Number	Description
2	WARNING: Insufficient space allocated for returning strings, strings truncated

C.4.6 CHANGIO - Get Parameters with Changed I/O Status:

C.4.6.1 Description: The general description of this function is applicable to the FORTRAN 77 implementation.

C.4.6.2 Function Definition:

```
INT FUNCTION CHANGIO (MAXPARM, NSTART, NUM, NAME, ID, IOSTAT,
          NUMERR, ERRVAL, ERRMSGS)

INTEGER MAXPARM, NUM, NSTART, ID(*), IOSTAT(*), NUMERR, ERRVAL(*)
CHARACTER*(*) NAME(*), ERRMSGS(*)
```

C.4.6.3 FORTRAN 77-Specific Error and Warning Messages:

TABLE C5 - FORTRAN 77-Specific Error and Warning Messages

Error/Warning Number	Description
3	WARNING: Insufficient space allocated for returning strings, strings truncated

C.4.7 SETCAT - Set Default Category:

C.4.7.1 Description: The general description of this function is applicable to the FORTRAN 77 implementation.