



GSP API 11 User Manual

GSP Development Team
www.gspteam.com

Version 11.4.8.0 27-Jul-16

GSP API 11 User Manual

by the GSP Development Team

NLR's Gas turbine Simulation Program (GSP) is an off-line component-based modeling environment for gas turbines. Both steady state and transient simulation of any kind of gas turbine configuration can be performed by establishing a specific arrangement of engine component models. GSP is a powerful tool for performance prediction and off-design analysis. GSP is especially suitable for sensitivity analysis of variables such as: ambient (flight) conditions, installation losses, certain engine malfunctioning (including control system malfunctioning), component deterioration and exhaust gas emissions.

© NLR 2016 - www.gspteam.com

GSP API 11 User Manual

© NLR 2016

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Publisher

*National Aerospace Laboratory NLR
Anthony Fokkerweg 2
1006 BM Amsterdam
The Netherlands*

Published

July 2016

Content editor

GSP Development Team

Information

www.gspteam.com



Table of Contents

1 Welcome	5
2 Document Conventions	7
3 Introduction	9
..1..Microsoft Windows DLL	9
..2..System requirements	9
..3..Documentation	10
..4..What's new?	10
..5..Limitations	11
..6..Authors	11
4 GSP standard API	15
...1..Introduction	15
...2..Flowcharts	15
.....Modeling process	15
.....Simulation process	16
...3..Model setup	16
...4..Model simulation	17
...5..GSP API details	17
.....Requirements	17
.....GSP API component library	17
.....DLL functions	19
.....About.....	19
.....CalculateDesignPoint.....	20
.....CalculateSteadyStatePoint.....	20
.....CloseModel.....	21
.....ConfigureModel.....	21
.....EditIterControl.....	21
.....EditTransControl.....	21
.....FreeAll.....	22
.....GetInputControlParameterArraySize.....	22
.....GetInputControlParameterByIndex.....	22
.....GetInputDataList.....	23
.....GetInputDataListSize.....	23
.....GetOutputDataArray.....	24
.....GetOutputDataList.....	24
.....GetOutputDataListSize.....	25
.....GetOutputDataParameterArraySize.....	25
.....GetOutputDataParameterValueByIndex.....	26
.....GetStStTable.....	26
.....GetStStTableSize.....	27
.....InitializeModel.....	27
.....InitTransient.....	27
.....LoadModel.....	28
.....LoadModelAnsi.....	28



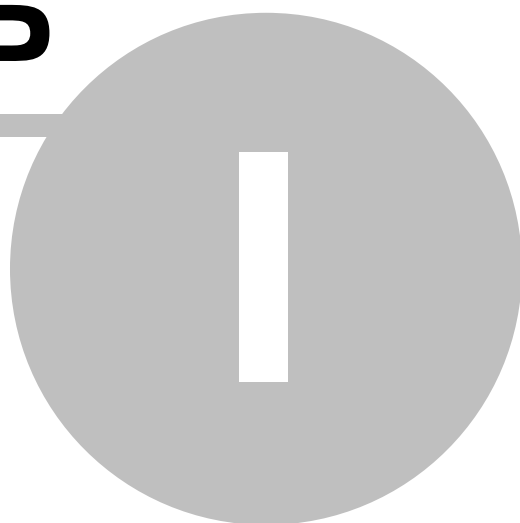
.....ModelLoaded.....	28
.....ProgramInfo.....	29
.....ResetODinputtoDP.....	29
.....RunModel.....	29
.....RunTransientStep.....	30
.....SaveModel.....	30
.....SetInputControlArray.....	30
.....SetInputControlParameterByIndex.....	31
.....SetTransientTimes.....	31
.....ShowConvergenceMonitor.....	31
.....ShowHideModel.....	32
.....Usage in MATLAB.....	32
.....Usage in SIMULINK.....	33
5 GSP ARP4868 API	36
...1...Aerospace Recommended Practice 4868	37
.....ARP 4868 function conventions.....	37
.....Function implementation in GSP API.....	38
...2...Function documentation	39
.....Additional standard ARP4868B function documentation	39
.....d4868initProg.....	39
.....d4868parseString.....	40
.....ExecFunction.....	41
.....GSPeditComponent.....	41
.....GSPGetNrOfComponents.....	42
.....GSPGetComponentNames.....	42
.....GSPeditOutputOptions.....	42
.....ExecProcedure.....	42
.....d4868terminate.....	42
.....Additional GSP specific functions.....	42
.....d4868writeStrToLog.....	43
.....d4868loadModel.....	43
.....d4868closeModel.....	44
.....d4868saveModel.....	45
.....d4868showModel.....	45
.....d4868getInputDataList.....	46
.....d4868getInputDataListSize.....	47
.....d4868programInfo.....	47
.....d4868evalFunction[IFDS].....	48
.....Functions for the GSP API developer	49
.....d4868initProgInitialized.....	49
.....d4868getAPIInterfaceComponent.....	49
.....d4868initFunction.....	50
.....d4868setFunctionError.....	50
.....d4868modelLoaded.....	50
.....d4868GetIndexOfOption.....	51
.....d4868GetValueOfOption.....	51
.....d4868listToString.....	51
.....d4868FreeAll.....	52
...3...Using the GSP ARP4868 API	52
.....Examples.....	55
.....Console application in C/C++.....	56
.....Console application in Delphi.....	63
...4...Error and warning messages	72
.....List of errors.....	72
.....List of warnings.....	74



6 Registration & Support	76
...1...Contact details	76
...2...Registration	76
.....Registration window	76
...3...Support from NLR	76
7 References	79
Index	80



GSP





1

Welcome

Gas turbine Simulation Program version 11 for MS-Windows

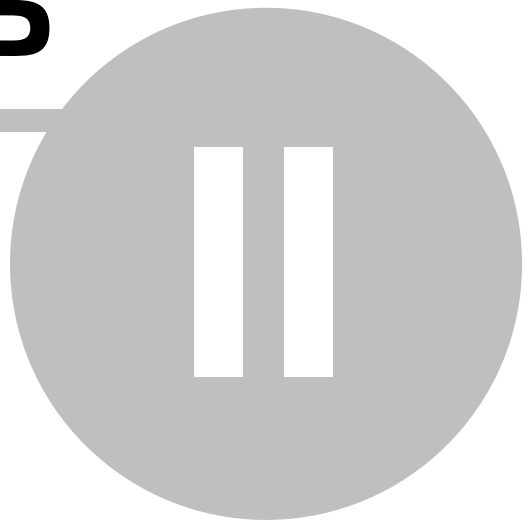
Authors [GSP Development Team members](#)
Copyrights [National Aerospace Laboratory NLR, Amsterdam](#)

Home page www.GSPteam.com

Help build nr 11.4.8.0
Help build date Wednesday, July 27, 2016



GSP



Document Conventions

Throughout the help guide the following conventions have been used:

- Normal font text is used to describe the help topics
This text is an example of the used font.
- Links to other topics are underlined and colored
[This link](#) shows an example. Note that the captions of the link does not always match the actual topic header for readability reasons.
- Fixed width font (courier) is used to indicate this is a name of a control of one of the controls used in either the GSP main application, the Project window, or any other input or data window.
This is an example of the font.
- Some options are found as sub options of option groups in e.g. file menus. The vertical line separator (|) is used to separate two or more possible options in command syntax.
File|Reopen|



GSP



3.1 Microsoft Windows DLL

Dynamic-link library (also written unhyphenated), or DLL, is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems. These libraries usually have the file extension DLL, OCX (for libraries containing ActiveX controls), or DRV (for legacy system drivers). The file formats for DLLs are the same as for Windows EXE files – that is, Portable Executable (PE) for 32-bit and 64-bit Windows, and New Executable (NE) for 16-bit Windows. As with EXEs, DLLs can contain code, data, and resources, in any combination.

DLL features:

- Since DLLs are essentially the same as EXEs, the choice of which to produce as part of the linking process is for clarity, since it is possible to export functions and data from either.
- It is not possible to directly execute a DLL, since it requires an EXE for the operating system to load it through an entry point, hence the existence of utilities like RUNDLL.EXE or RUNDLL32.EXE which provide the entry point and minimal framework for DLLs that contain enough functionality to execute without much support.
- DLLs provide a mechanism for shared code and data, allowing a developer of shared code/data to upgrade functionality without requiring applications to be re-linked or re-compiled. From the application development point of view Windows and OS/2 can be thought of as a collection of DLLs that are upgraded, allowing applications for one version of the OS to work in a later one, provided that the OS vendor has ensured that the interfaces and functionality are compatible.
- DLLs execute in the memory space of the calling process and with the same access permissions which means there is little overhead in their use but also that there is no protection for the calling EXE if the DLL has any sort of bug.

3.2 System requirements

GSP 11 runs on MS-Windows operating systems with the following configurations:

	Minimum
<i>Operating System</i>	Windows XP
<i>Processor</i>	Pentium III
<i>RAM Memory</i>	128 MB
<i>Resolution</i>	SVGA 1024x768 @ 256+ colors
	Recommended
<i>Operating System</i>	7, 8 (32-bit or 64-bit)
<i>Processor</i>	Intel or AMD 2 GHz processor or faster
<i>RAM Memory</i>	1 GB
<i>Resolution</i>	SVGA 1280x1024 or higher @ 256+ colors

Note that the GSP API itself does not require minimum requirements for the screen resolution since the engine models can be run non-visually (running in the background).

3.3 Documentation

Public printed documentation is provided in:

- *GSP User Manual*
The User Manual primarily provides information necessary to use the program for running simulations on existing gas turbine models. The User Manual is the printed version of the on-line help.
- *GSP Technical Manual* ([registered](#) users only)
The GSP Technical Manual provides detailed information on the thermodynamics and numerical mathematics applied in the simulation environment and component models. This manual is required for more advanced use, including the development of new gas turbine models and new component models.
- *Several publications presented at conferences* (listed in [References](#))

For the Component Developers Package (CDP) license there further is the

- *GSP Component Model Developer's Manual*
The GSP Component Model Developers Manual is for developing new custom component models and requires the Borland Delphi software development environment and the GSP Component Developers Package

Additional documentation not publicly available includes:

- *Software Requirements Specification (SRS) for GSP*
- *GSP Analysis and Design Document*

Documentation can be obtained from the NLR GSP site. For additional information [contact NLR](#).

3.4 What's new?

```

NLR GAS TURBINE SIMULATION PROGRAM GSP API for MS-WINDOWS          RELEASE NOTES
-----

=====
GSP v11.3.4.0                                                         10-12-2013
-----
* Beta release for API testers. (O. Kogenhop NLR)

GSP v11.3.3.1                                                         29-11-2013
-----
Improvements
-----
* Added a scale value column to the output parameter grid of the API interface
  component. (O. Kogenhop NLR)

New features
-----
* Added registration keys for registering the GSP API version. (O. Kogenhop NLR)
* Free GSP API (Microsoft Windows DLL) version included in the installer.
  (O. Kogenhop NLR)
* New and updated GSP API (DLL) functions to use GSP from other software
  programs. (O. Kogenhop NLR)

=====

```

3.5 Limitations

Do not load multiple model files with the GSP API.

Opening an additional model while a model is loaded with a single instance of the GSP API will result in closing the opened model first.

Do not start multiple instances of the GSP API as you will run into errors stating that you cannot open data bases as they are in use.

3.6 Authors

GSP is continuously being developed since 1996 by a steadily growing group of authors and software developers representing the *GSP Development Team* or "*GSP Team*" of www.gspteam.com. The GSP Team includes members from NLR, Delft University of Technology and users/developers from several institutes and industries. The coordination of the development work is performed by NLR and Delft University. Below, the authors of the main GSP elements are listed (see also [References](#)):

(updated November 2013)

Main program

- Kernel, object oriented architecture and design (1996-1998)
Wilfried P.J. Visser
- Main program, standard component library and GUI
Oscar Kogenhop, Wilfried P.J. Visser, Michael Broomhead
- Case management architecture and XML storage and inheritance mechanism
Wilfried P.J. Visser, Oscar Kogenhop
- Generic control system component library :
Wilfried P.J. Visser, Oscar Kogenhop, Michael Broomhead
- Parametric P3T3 emission models in combustor module
Michiel Bruin
- Gas and combustion model for user specified fuels
Wilfried P.J. Visser
- Multi-reactor emission models in combustor module
Steven Kluiters, Wilfried P.J. Visser
- Thermal network simulation functionality / heat sink component
Wilfried P.J. Visser

Component Developers Package

- A special source code release for customer in-house development
*Oscar Kogenhop,
Wilfried P.J. Visser*

Application Programming Interface

- Development of a windows dll file containing functions to simulate GSP models from other software or programming languages
*Oscar Kogenhop,
Wilfried P.J. Visser*

- Development of an MATLAB-SIMULINK S-function to allow usage of GSP in SIMULINK (C-code development)

*Erik H. Baalbergen,
Oscar Kogenhop*

Application specific libraries

- Additional libraries with:
 - custom control system components
 - custom heat exchanger components
 - custom pressure vessel inlet for turbine powered wind tunnel engine simulator model
 - customer specific components

*Oscar Kogenhop,
Wilfried P.J. Visser,
Edward R. Rademaker*

- STOVL specific components

*Wilfried P.J. Visser
Michael Broomhead*

- Supplementary components

*Oscar Kogenhop,
Michael Broomhead,
Wilfried P.J. Visser*

- Gas path analysis / Adaptive modeling library

*Wilfried P.J. Visser
Oscar Kogenhop
Mark Oostveen*

- TUD / KLM Gas path analysis / Adaptive modeling library

*Wilfried P.J. Visser
Michel Verbist*

- MTT micro turbine component library

Wilfried P.J. Visser

- NLR component library

Oscar Kogenhop

Documentation

- GSP User Manual / On-line help

*Oscar Kogenhop
Wilfried P.J. Visser
Michael Broomhead*

- GSP Technical Manual

*Oscar Kogenhop
Wilfried P.J. Visser
Michael Broomhead
Edward R. Rademaker*



- GSP Component Developers Package (CDP) Manual
Oscar Kogenhop
Michael J. Broomhead
Michiel J.D. Valens
Wilfried P.J. Visser
- GSP API Manual
Oscar Kogenhop
- Heat transfer modeling
Michel Verbist



GSP

IV

4 GSP standard API

This section describes the use of the GSP standard API.

4.1 Introduction

The GSP API extends the use of the GSP stand alone program by enabling use of the simulation capabilities to be run outside the main GSP stand alone program.

The GSP API has been explicitly designed for use in the generic modeling environment of MATLAB and MATLAB SIMULINK. Both MATLAB and SIMULINK are registered trademarks owned by the MathWorks simulation software company.

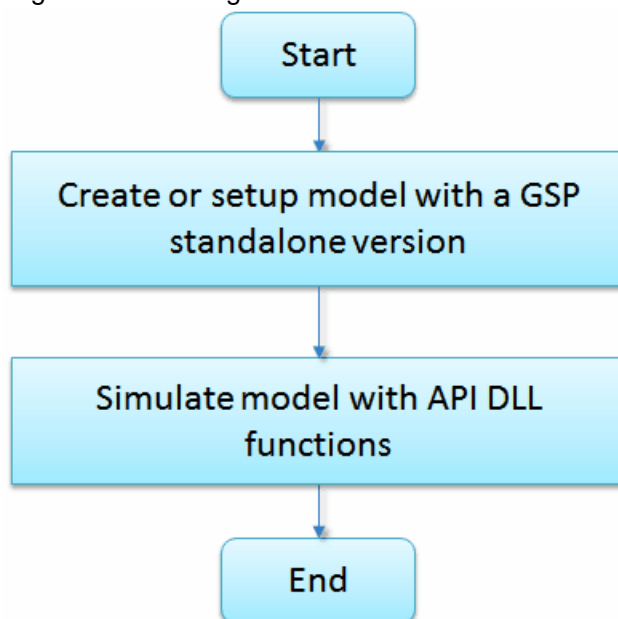
Note that the API has been developed as a 32-bit (x86) Microsoft DLL library implying that it can be used from any other 32-bit program running on any Windows platform (both 32-bit and 64-bit versions of Windows).

4.2 Flowcharts

The processes for [simulation](#) and the actual [simulation](#) are described to give a clear overview of how the API should be used.

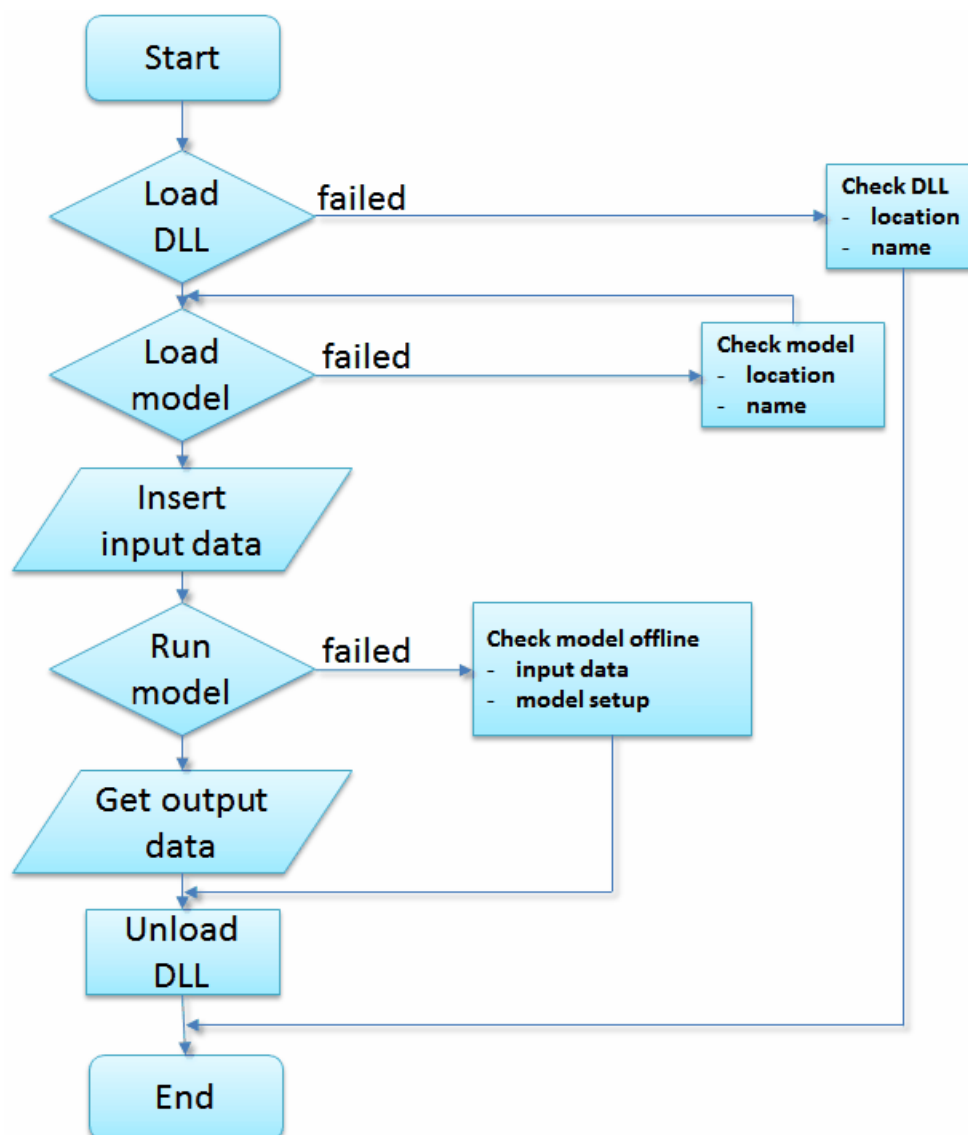
4.2.1 Modeling process

The flowchart below shows a simplified, graphical representation of the process to simulate engine models using the GSP API.



4.2.2 Simulation process

The flowchart below shows a simplified, graphical representation of the simulation process using the GSP API functions.



4.3 Model setup

The API version is an extension of the GSP standalone program. Basically all GSP models can be simulated with the API. By introducing a new input component model, every GSP model can be changed into an API model for simulation with the DLL functions.

This input component model has been described in the section [GSP API component library](#) in detail.

Basically, the modeler needs to specify the input parameters for the model and the output parameters that need to be obtained from the model. Save this model, do not change the case before saving! The API will use the case model that is the active case!

To minimize the interfacing with the model during simulation, it is advised to set all model options such that few error messages are displayed to the modeler. The DLL function

[ConfigureModel](#) can be used to do this automatically for you.

4.4 Model simulation

Running models from other software should conform to the specified order in this section.

The simulation process is shown in flowchart [Simulation process](#).

First, the DLL library has to be loaded into memory. Loading a DLL into memory is described by the environment you are calling the DLL from. E.g. in C-code the function `LoadLibrary()` can be used to load the DLL. In MATLAB the function is called `loadlibrary()`.

Note that for accessing the functions in the library, the prototypes of the functions are required. Without knowing these prototypes, function cannot be found in the DLL, and the amount and type of the arguments is unknown to call the

4.5 GSP API details

This section contains information on the [GSP API component library](#) and the individual [DLL functions](#).

4.5.1 Requirements

The DLL currently only supports win32. This requires that a 32-bit application (no problem if it is running on a 64-bit platform as long it is not run by a 64-bit program) is used to call the DLL from.

4.5.2 GSP API component library

The API interface component is a component derived from the case input controllers. The component provides the necessary interface to define the input to the gas turbine model and the output from the gas turbine model.

The API interface component can be found in the "API" library:

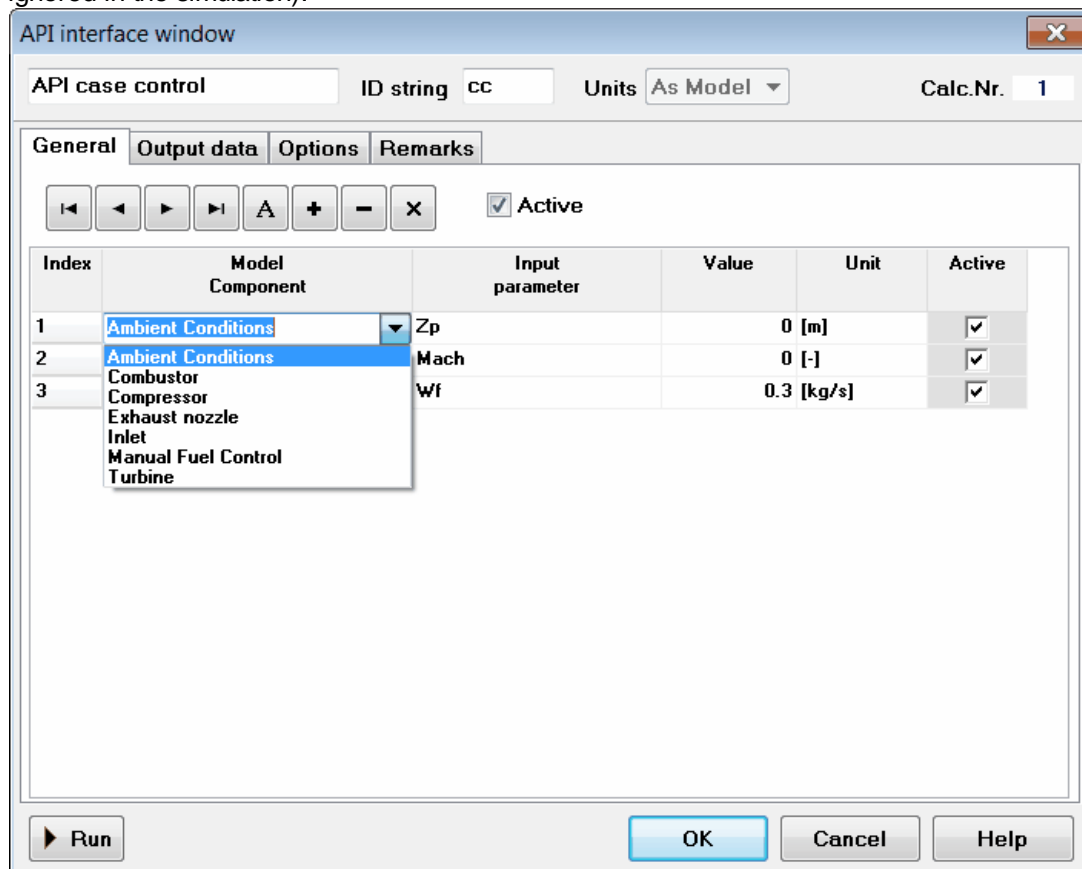


Add this component to the case model you would like to simulate with the API. It is essential that the run mode/case type is set to *Steady-State* when setting up the component. This enables the modeler to have control over all the off-design input parameters. Note that for design analyses the case type should be set to *design*!

The component requires that the modeler specifies the input and output parameters for the cycle.

The `General` tab sheet shows the input parameter grid. By selecting a model component

from the drop down list in the `Model component` column, a drop down list with input parameters (if available for that component!) will be accessible in the `Input parameter` column. Upon selecting the input parameter, the current value and the unit caption will be copied into the respective columns. The checkbox in the `Active` column is used to (de)activate the input parameter (i.e. if unselected, the value of the input parameter will be ignored in the simulation).



The screenshot shows the 'API interface window' with the 'General' tab selected. The window has a title bar with a close button. Below the title bar, there are fields for 'API case control', 'ID string' (set to 'cc'), 'Units' (set to 'As Model'), and 'Calc.Nr.' (set to '1'). Below these fields are four tabs: 'General', 'Output data', 'Options', and 'Remarks'. The 'General' tab is active and contains a set of navigation buttons (left, right, first, last, search, add, subtract, multiply) and a checked 'Active' checkbox. Below the navigation buttons is a table with the following data:

Index	Model Component	Input parameter	Value	Unit	Active
1	Ambient Conditions	Zp	0	[m]	<input checked="" type="checkbox"/>
2	Ambient Conditions	Mach	0	[-]	<input checked="" type="checkbox"/>
3	Compressor	Wf	0.3	[kg/s]	<input checked="" type="checkbox"/>

A dropdown menu is open for the 'Model Component' column, showing the following options: Ambient Conditions, Combustor, Exhaust nozzle, Inlet, Manual Fuel Control, and Turbine. At the bottom of the window are four buttons: 'Run', 'OK', 'Cancel', and 'Help'.

The `Output data` tab sheet shows the output parameter grid. By selecting an output parameter from the drop down list in the `Parameter` column, a drop down list with output parameters (these are all the output parameters that have been selected in the output tab sheets of all components supplemented with system output parameters) will be shown. By selecting an output parameter, the current value and the unit caption will be copied into the respective columns. The checkbox in the `Enabled` column is used to (de)activate the output parameter (i.e. if unselected, the value of the output parameter will not be passed through to the API).

API interface window

API case control ID string cc Units As Model Calc.Nr. 1

General Output data Options Remarks

File logging

☐ Write output to file during simulation

GSPoutput.txt

⏮ ⏪ ⏩ ⏭ A + - X

Enabled	Parameter	Value	Unit
<input checked="" type="checkbox"/>	FN	11.5067149582	[kN]
<input checked="" type="checkbox"/>	TSFC	0.0938582387698	[kg/N h]

PWshaft_t
SM_c
TQ_c
TQ_t
TS9
TSFC
TT1
TT2

Run OK Cancel Help

4.5.3 DLL functions

All public [DLL](#) functions will be described what the purpose of the function is.

All functions return a Boolean value. This value indicates if the function has or has not succeeded in performing the required action. Input to and output from the function is regulated through the function parameters.

Note that the DLL needs to be loaded into memory before the functions can be used.

4.5.3.1 About

Function description:

Displays the "About" window to show the GSP version information.

Function prototype:

```
bool __cdecl About ( ) ;
```



4.5.3.2 CalculateDesignPoint

Function description:

This function will perform a design calculation. This is required for GSP to size the engine and calculate relevant design parameters from the user input.

Function prototype:

```
bool __cdecl CalculateDesignPoint(bool Do_Output, bool
AlwaysErrorOutput, bool NoConfirmOutput);
```

Parameters:

- **Do_Output**
Boolean input parameter to specify whether output will be generated (if `true`) in the steady state output table.
- **AlwaysErrorOutput**
Boolean input parameter to produce output results (if `true`) when an error has occurred. Note that this can be erroneous, non converged data.
- **NoConfirmOutput**
Boolean input parameter that indicates whether output will be written to the table without showing a message dialog (if `true`).

4.5.3.3 CalculateSteadyStatePoint

Function description:

This function will perform a steady state calculation. The user requires to set the steady state input data prior to running this function using functions `SetInputControlArray` or preferred: `SetInputControlParameterByIndex`.

Function prototype:

```
bool __cdecl CalculateSteadyStatePoint(bool DoShowProgBox, bool
DoTableOutput);
```

Parameters:

- **DoShowProgBox**



Boolean input parameter to specify whether iteration progress box will be shown on the screen (if `true`).

- `DoTableOutput`
Boolean input parameter to specify whether output will be generated (if `true`) in the steady state output table.

4.5.3.4 CloseModel

Function description:

Closes the open/active model.

Function prototype:

```
bool __cdecl CloseModel(bool NoSaveDlg);
```

Parameters:

- `NoSaveDlg`
This function uses a Boolean input parameter that, if omitted, (default = `false`) or `false` closes the model if model changes are detected with showing the save dialogs. When `true`, the save dialog is not shown and the model will be closed without user input.

4.5.3.5 ConfigureModel

Function description:

Use this function to automatically configure the model such that the model options are configured optimally for use with the API (i.e. no table pop-up for output data, reports, etc).

Function prototype:

```
bool __cdecl ConfigureModel();
```

Parameters:

- none

4.5.3.6 EditIterControl

Function description:

Opens the model's iteration control parameters options window for editing.

Function prototype:

```
bool __cdecl EditIterControl();
```

Parameters:

- none

4.5.3.7 EditTransControl

Function description:

Opens the model's Transient/St.St.series control parameters options window for editing.

Function prototype:

```
bool __cdecl EditTransControl();
```

Parameters:

- none

4.5.3.8 FreeAll

Function description:

Free the loaded windows (get loaded when a DLL file is loaded into memory), this has to be done prior to unloading a DLL for proper memory management.

Function prototype:

```
bool __cdecl FreeAll();
```

Parameters:

- none

4.5.3.9 GetInputControlParameterArraySize

Function description:

This function will retrieve the amount of the input control parameters defined in input parameter grid of the API interface component.

Function prototype:

```
bool __cdecl GetInputControlParameterArraySize(int *InputArraySize);
```

Parameters:

- InputArraySize
Pointer to integer argument; the function will return the amount of parameters in the input parameter grid of the [GSP API input component](#).

4.5.3.10 GetInputControlParameterByIndex

Function description:

Function will get the name and value of an input parameter by the index of the input parameter in the input parameter grid.

Function prototype:

```
bool __cdecl GetInputControlParameterByIndex(int ParamIndex, char *ParamName, double *ParamValue);
```

Parameters:

- ParamIndex
This integer input parameter specifies the index of the input control parameter defined (the first control parameter has index 1). Note that the index corresponds to the row number (the column header/title is row 0).
- ParamName
This parameter returns the name of the input parameter as used in GSP. This string includes the component name: e.g. "ManualFuelControl.Wf".
- ParamValue
This double output parameter returns the current value of the input parameter specified in the input parameter grid.

4.5.3.11 GetInputDataList

Function description:

Function to retrieve the amount of input parameters, input values, units and the row index from the model. Note that the `char **` need to be correctly allocated prior to running this function (e.g. use the [GetInputDataListSize](#) function to get the maximum length of the largest string in the list, the longest possible unit string is about 11, unless custom defined units are used).

Function prototype:

```
bool __cdecl GetInputDataList(char **paramList, double
*valueList, char **unitList, int *indexList, int size);
```

Parameters:

- `paramList`
In this parameter (pointer to a pointer to character or basically an array) the list of active parameters from the input parameter grid of the [GSP API input component](#) is retrieved.
- `valueList`
This parameter returns the values of the all the active input parameters.
- `unitList`
In this parameter (pointer to a pointer to character or basically an array) the list of active parameters unit strings from the input parameter grid of the [GSP API input component](#) is retrieved.
- `indexList`
This integer input parameter specifies the index of the input control parameter defined (the first control parameter has index 1). Note that the index corresponds to the row number (the column header/title is row 0). As there are active and inactive parameters the index can be used to determine the place in the input grid for setting a value using function [SetInputControlParameterByIndex](#).
- `size`
This integer input parameter determines the amount of parameters are returned (use function [GetInputDataListSize](#) to get this value).

4.5.3.12 GetInputDataListSize

Function description:

This function will retrieve the amount of input parameters from the model. The amount will be returned in `listSize`. Note that the length of the longest string is returned in `maxStringSize` (unless `maxStringSize` equals the null pointer). The `listSize` and `maxSizeString` are required before the [GetInputDataList](#) function can be called.

Function prototype:

```
bool __cdecl GetInputDataListSize(int *listSize, int
*maxSizeString);
```

Parameters:

- `listSize`
Pointer to integer argument; the function will return the amount of active parameters in the input parameter grid of the [GSP API input component](#).
- `maxSizeString`
Pointer to integer argument; the function will return the largest string size of all the

components in the input parameter grid of the [GSP API input component](#).

4.5.3.13 GetOutputDataArray

Function description:

This function gets an array of output parameter values from the output parameter grid where each array element will be stored at the corresponding row (array element 1 will be stored at row 1, etc.). The input is a Delphi array. Note that Delphi arrays are incompatible with array definitions in C (only referring to the first element). Mapping Delphi arrays in C(++)-code requires the use of extra coding, by mimicking the Delphi memory storage in C-code. Therefore the function below cannot be called directly! If so, the DLL and thus the program calling the DLL will crash!

NOTICE: Usage of this function is **strongly discouraged** without proper understanding of Delphi and C memory management.

The function argument is a pointer to a character, which points to a certain location in the memory that has been designed to store an array in Delphi style. Instead of using this function it is advised to code a loop using function [GetOutputDataParameterValueByIndex](#).

Function prototype:

```
bool __cdecl GetOutputDataArray(char *OutputValArray);
```

Parameters:

- OutputValArray
Pointer to character argument; this parameter will be used to obtain an array of doubles from the model.

Note that the character pointer points to an address in memory where the Delphi array has to be mimicked. This requires additional coding in the host application. This code development is not supported by the GSP development team. This function, along with the additional Delphi array coding in C, is used in the interface code of the MATLAB SIMULINK S-function and available as a binary file (MATLAB MEX file).

4.5.3.14 GetOutputDataList

Function description:

Function to retrieve the amount of output parameters, input values, units and the row index from the model. Note that the `char **` need to be correctly allocated prior to running this function (e.g. use the [GetOutputDataListSize](#) function to get the maximum length of the largest string in the list, the longest possible unit string is about 11, unless custom defined units are used).

Function prototype:

```
bool __cdecl GetOutputDataList(char **paramList, double *valueList, char **unitList, int *indexList, int size);
```

Parameters:

- paramList
In this parameter (pointer to a pointer to character or basically an array) the list of active parameters from the output parameter grid of the [GSP API input component](#) is retrieved.
- valueList
This parameter returns the values of the all the active input parameters.
- unitList



In this parameter (pointer to a pointer to character or basically an array) the list of active parameters unit strings from the output parameter grid of the [GSP API input component](#) is retrieved.

- **indexList**
This integer input parameter specifies the index of the output control parameter defined (the first control parameter has index 1). Note that the index corresponds to the row number (the column header/title is row 0). As there are active and inactive parameters the index can be used to determine the place in the input grid for setting a value using function [SetInputControlParameterByIndex](#).
- **size**
This integer input parameter determines the amount of parameters are returned (use function [GetOutputDataListSize](#) to get this value).

4.5.3.15 GetOutputDataListSize

Function description:

This function will retrieve the amount of output parameters from the model. The amount will be returned in `listSize`. Note that the length of the longest string is returned in `maxStringSize` (unless `maxStringSize` equals the null pointer). The `listSize` and `maxSizeString` are required before the [GetOutputDataList](#) function can be called.

Function prototype:

```
bool __cdecl GetOutputDataListSize(int *listSize, int
*maxSizeString);
```

Parameters:

- **listSize**
Pointer to integer argument; the function will return the amount of active parameters in the output parameter grid of the [GSP API input component](#).
- **maxSizeString**
Pointer to integer argument; the function will return the largest string size of all the components in the output parameter grid of the [GSP API input component](#).

4.5.3.16 GetOutputDataParameterArraySize

Function description:

Retrieves the amount of the output control parameters defined in the output parameter grid inside the API interface component.

Function prototype:

```
bool __cdecl GetOutputDataParameterArraySize(int
*OutputArraySize);
```

Parameters:

- **OutputArraySize**
Pointer to integer argument; the function will return the amount of parameters in the parameter.

4.5.3.17 GetOutputDataParameterValueByIndex

Function description:

The function gets the value of the specified output parameter defined in the output parameter grid of the API interface component by the index (row nr.) in the output data grid.

Function prototype:

```
bool __cdecl GetOutputDataParameterValueByIndex(int ParamIndex,
double *ParamValue, bool DoScale);
```

Parameters:

- ParamIndex
This integer input parameter specifies the index (row nr.) of the output parameter defined in the output grid (the first control parameter has index 1). Note that the index corresponds to the row number (the column header/title is row 0, the first output parameter is located at row = 1).
- ParamValue
This pointer to a double refers to a variable which will get the value of the output parameter referred by the index in the output parameter grid of the API interface component.
- DoScale
This Boolean parameter is *false* by default, and can therefore be omitted. The function returns the scaled value (to the unit system set in the model) when the parameter is *true*. E.g. if FN (net Thrust) has been selected as an output parameter, the function will return a value in [kN] in the parameter ParamValue if the DoScale is omitted or false; if the DoScale value is true, then the ParamValue will be in [N].

4.5.3.18 GetStStTable

Function description:

This function stores the steady state table in a predefined (allocated) char array that is passed by a pointer to a char pointer.

Function prototype:

```
bool __cdecl GetStStTable(char **StStReport, int maxLines, int
maxSizeString);
```

Parameters:

- StStReport
In this parameter (pointer to a pointer to character or basically an array) the steady state table will be stored as text.
- maxLines
This parameter inputs the maximum allocated lines in StStReport. Basically this parameter together with maxSizeString is used to allocate the memory of StStReport.
- maxSizeString
This parameter inputs the maximum string size in StStReport. Basically this parameter together with maxLines is used to allocate the memory of StStReport.

4.5.3.19 GetStStTableSize

Function description:

This function retrieves the amount of characters that need to be allocated to store the steady state output table to an array of strings.

Function prototype:

```
bool __cdecl GetStStTableSize(int *maxLines, int *maxSizeString);
```

Parameters:

- `maxLines`
This parameter outputs the maximum allocated lines in `StStReport`. Basically this parameter together with `maxSizeString` must be used to allocate the memory of the report.
- `maxSizeString`
This parameter outputs the maximum string size in `StStReport`. Basically this parameter together with `maxLines` must be used to allocate the memory of the report.

4.5.3.20 InitializeModel

Function description:

This function will initialize the model.

Function prototype:

```
void __cdecl InitializeModel();
```

Parameters:

- none

4.5.3.21 InitTransient

Function description:

This function will initialize the transient calculation run mode, this function must be called once prior to the start of the beginning of the series of transient calculations. Basically, this procedure sets and initializes various parameters to be able to do transient analyses (set initial start time, end time, time step, output interval, various counters, etc.).

Function prototype:

```
bool __cdecl InitTransient(bool StabilizedDlg, bool Stabilize,  
bool ShowProgrBox, double TransStepStartTime, double TimeStep);
```

Parameters:

- `StabilizedDlg`
Controls whether or not the stabilize dialog is shown (if `true`) to the modeler.
- `Stabilize`
Indicates whether the simulation is required to stabilize to the current input parameters (without showing the dialog).
- `ShowProgrBox`
Controls whether or not to display the progress box. This should be false as the end time is not known before hand when controlled from other software.
- `TransStepStartTime`
Initial start time.

- `TimeStep`
Initial value for the time step (`dt`). Note that for variable solvers, the time step can be overwritten at a later stage.

4.5.3.22 LoadModel

Function description:

Load a model from the given file name and path. When a model has been loaded before this model, first the loaded model will be closed, and the `ModelFileName` model loaded. I.e. only one model can be loaded. This function is the UniCode version (supporting wide characters) of the [LoadModelAnsi](#) function.

Function prototype:

```
bool __cdecl LoadModel(WCHAR *ModelFileName, bool ShowModel);
```

Parameters:

- `ModelFileName`
A UniCode string to define the file name and path of the model that needs to be loaded.
- `ShowModel`
A Boolean input parameter controlling to show the model or not.
If `true`, the model will be shown upon loading. If `false`, the model is hidden, only the loading progress indicator is shown.

4.5.3.23 LoadModelAnsi

Function description:

See [LoadModel](#), this function only differs in the parameter type `ModelFileName`.

Function prototype:

```
bool __cdecl LoadModel(char *ModelFileName, bool ShowModel);
```

Parameters:

- `ModelFileName`
An Ansi type string to define the file name and path of the model that needs to be loaded.
- `ShowModel`
A Boolean input parameter controlling to show the model or not.
If `true`, the model will be shown upon loading. If `false`, the model is hidden, only the loading progress indicator is shown.

4.5.3.24 ModelLoaded

Function description:

Check if a model is loaded

Function prototype:

```
bool __cdecl ModelLoaded(bool ShowMessage, bool ShowNotLoadedError);
```

Parameters:

- `ShowMessage`
A Boolean input parameter controlling to show a message.
If `true`, a message will always be reported/shown whether the model is loaded or not.
- `ShowNotLoadedError`
A Boolean input parameter controlling to show a message only when no model is loaded.



If `true` and no model is loaded, an error message is generated/shown. Note that this parameter is only active if `ShowMessage = false`.

4.5.3.25 ProgramInfo

Function description:

Returns basic program information.

Function prototype:

```
bool __cdecl ProgramInfo(char *Name, char *Version);
```

Parameters:

- `Name`
Returning a character pointer to the name of the program.
- `Version`
Returning a character pointer to the version string of the program.

4.5.3.26 ResetODinputtoDP

Function description:

Resets the OD input data to DP values (this cannot be reverted).

Function prototype:

```
void __cdecl ResetODinputtoDP(bool DoConfirm);
```

Parameters:

- `DoConfirm`
This function uses a Boolean input parameter that, if omitted, (default = `true`) or `true` will show a confirmation dialog whether the user wants to override the current input data as it is about to overwrite the OD input data by the respective design input data. When `false`, the function works silently by executing the reset without user interference.

4.5.3.27 RunModel

Function description:

Run the case model using the case type defined in the model file.

Function prototype:

```
bool __cdecl RunModel(bool StartTimeDlg, bool StabilizedDlg, bool Stabilize, bool ShowProgrBox);
```

Parameters:

- `StartTimeDlg`
Boolean input parameter to show (`true`) or hide (`false`) the start time dialog.
- `StabilizedDlg`
Boolean input parameter to show (`true`) or hide (`false`) the stabilize dialog.
- `Stabilize`
Boolean input parameter to stabilize the simulation (`true`) at the current time. I.e. a steady state calculation for the current input conditions will be calculated.
- `ShowProgrBox`
Boolean input parameter to show (`true`) or hide (`false`) the progress bar window.

4.5.3.28 RunTransientStep

Function description:

This function will calculate the simulation results for a single transient step for a specified time using a specified time step.

Note that the time step may differ from the time step defined in [InitTransient](#).

Function prototype:

```
bool __cdecl RunTransientStep(bool ShowProgrBox, double
TransStepStartTime, double TimeStep);
```

Parameters:

- ShowProgrBox
Controls whether or not to display the progress box. This should be false as the end time is not known before hand when controlled from other software.
- TransStepStartTime
Start time of the integration interval.
- TimeStep
Value for the time step (dt)/integration time.

4.5.3.29 SaveModel

Function description:

Saves the [loaded](#) model.

Function prototype:

```
bool __cdecl SaveModel(bool SaveAsDlg);
```

Parameters:

- StartTimeDlg
Boolean input parameter to show (true) a save dialog window upon saving. If false, model is saved instantly with the current file name.

4.5.3.30 SetInputControlArray

Function description:

This function sets an array of input parameter values in the input parameter grid where each array element will be stored at the corresponding row (array element 1 will be stored at row 1, etc.). The input is a Delphi array. Note that Delphi arrays are incompatible with array definitions in C (only referring to the first element). Mapping Delphi arrays in C(++)-code requires the use of extra coding, by mimicking the Delphi memory storage in C-code. Therefore the function below cannot be called directly! If so, the DLL and thus the program calling the DLL will crash!

NOTICE: Usage of this function is **strongly discouraged** without proper understanding of Delphi and C memory management.

The function argument is a pointer to a character, which points to a certain location in the memory that has been designed to store an array in Delphi style. Instead of using this function it is advised to code a loop using function [SetInputControlParameter](#).

Function prototype:

```
bool __cdecl SetInputControlArray(char *InputValArray);
```

Parameters:

- InputValArray

Pointer to character argument; this parameter will be used to pass an array of doubles to the model.

Note that the character pointer points to an address in memory where the Delphi array has to be mimicked. This requires additional coding in the host application. This code development is not supported by the GSP development team. This function, along with the additional Delphi array coding in C, is used in the interface code of the MATLAB SIMULINK S-function and available as a binary file (MATLAB MEX file).

4.5.3.31 SetInputControlParameterByIndex

Function description:

Function will set the value of an input parameter by the index of the input parameter in the input parameter grid.

Function prototype:

```
bool __cdecl SetInputControlParameterByIndex(int ControlVarIndex,  
double ControlValue);
```

Parameters:

- **ControlVarIndex**
This integer input parameter specifies the index of the defined input control parameter (the first control parameter has index 1). Note that the index corresponds to the row number (the column header/title is row 0).
- **ControlValue**
This double input parameter specifies the value of the input parameter.

4.5.3.32 SetTransientTimes

Function description:

Function will set the begin and end time for a transient simulation. This has to be called prior to running the transient simulation.

Function prototype:

```
bool __cdecl SetTransientTimes(double StartTime, double  
MaxEndTime);
```

Parameters:

- **Starttime**
Specify to set the current transient start time (TransTime) value, which is the initial transient/steady-state series time value (i.e. the starting time for the transient calculation).
- **MaxEndTime**
Only used if larger than zero: it then replaces the model's Transient/Series Options setting, which is saved with model settings ! if smaller than zero, then the simulation stop/end time is taken from existing Transient/Series Options 'maximum time' setting (1E20 seconds).

4.5.3.33 ShowConvergenceMonitor

Function description:

Function to show and hide the "Convergence monitor". The convergence monitor window graphically displays the error convergence during simulations.

Function prototype:

```
bool __cdecl ShowConvergenceMonitor(bool DoShow);
```

Parameters:

- DoShow
Boolean parameter to show (True) or hide (False) the convergence monitor window.

4.5.3.34 ShowHideModel

Function description:

Can be used for showing and hiding the model window

Function prototype:

```
bool __cdecl ShowHideModel(bool ShowModel);
```

Parameters:

- ShowModel
Boolean input parameter to show (true) or hide (false) the model that has been previously loaded with the GSP API function [LoadModel](#).

4.5.4 Usage in MATLAB

Requirements:

- MATLAB/SIMULINK
 - 32-bit MATLAB (64-bit is not supported yet)
 - Select a C(++) compiler with command `mex -setup`
 - Select the Microsoft Software Development Kit (SDK) 7.1 (this must be installed first, free download from Microsoft.com)
- The following files in a single directory (PATH added to MATLAB):
 - GSP.dll
 - GSP.h
 - GSPAPI.cfg
This file defines the amount of input and output parameters and the model file name.
 - YourModelFileName.mxl
- The stand-alone version must be closed!

Example of the using the GSP API in MATLAB:

1. Configure a model and prepare it for usage with the API (see [GSP API component library](#)). Close GSP stand alone version when finished.
2. Place the following files in a single folder:
 - YourModelFileName.mxl
 - GSP.dll
 - GSP.h
3. Open MATLAB and navigate to the folder containing the model and GSP API files. Add the path to the MATLAB environment.
4. Execute the following commands:


```
>> loadlibrary('GSP.dll','GSP.h','alias','gspdll'); % Loads the
GSP API DLL in memory setting an alias called 'gspdll'
>> calllib('gspdll','About'); % Optional, shows a window with
version information
>> calllib('gspdll','LoadModelAnsi','YourModelFileName.mxl',0); %
Load <YourModelFileName>.mxl
>> calllib('gspdll','CalculateDesignPoint',0,0,1); % Optional,
will be done automatically when model has not been run prior to
'RunModel'
```

```
>> calllib('gspdll','SetInputControlParameterByIndex',3,0.38); %  
Set e.g. the fuel flow (parameter on row 3) Wf to 0.38 kg/s  
>> calllib('gspdll','RunModel',0,0,0,0); % Run the currently  
defined run case type  
>> dv=0.00; % Create a double variable and initialize to zero  
>> pdv = libpointer('doublePtr',dv); % Now create a pointer to  
this double variable  
>> calllib('gspdll','GetOutputDataParameterValueByIndex',1,  
pdv,0); % Get the thrust value using a pointer to a double  
parameter  
>> pdv.Value  
  
ans =  
  
14.1646 % This is the thrust in kN  
  
>> calllib('gspdll','CloseModel',1); % Close the model, no save  
dialog, no saving  
>> calllib('gspdll','FreeAll'); % Free allocated memory  
>> unloadlibrary gspdll; % Finally, unload the API to release the  
DLL file from memory
```

4.5.5 Usage in SIMULINK

Requirements:

- MATLAB/SIMULINK
 - 32-bit MATLAB (64-bit is not supported yet)
 - Select a C(++) compiler with command `mex -setup`
 - Select the Microsoft Software Development Kit (SDK) 7.1 (this must be installed first, free download from Microsoft.com)
- The following files in a single directory (PATH added to MATLAB):
 - GSP.dll
 - GSP.h
 - GSPAPI.cfg
This file defines the amount of input and output parameters and the model file name.
 - YourModelFileName.mxl
 - GSPAPI.mexw32
Note that the correct version has been copied! This should correspond to the MATLAB version and the software platform version (currently only 32-bit is supported). A debug and a release version are supplied, the difference is that the debug version outputs more hints in the log file.
- The stand-alone version must be closed!

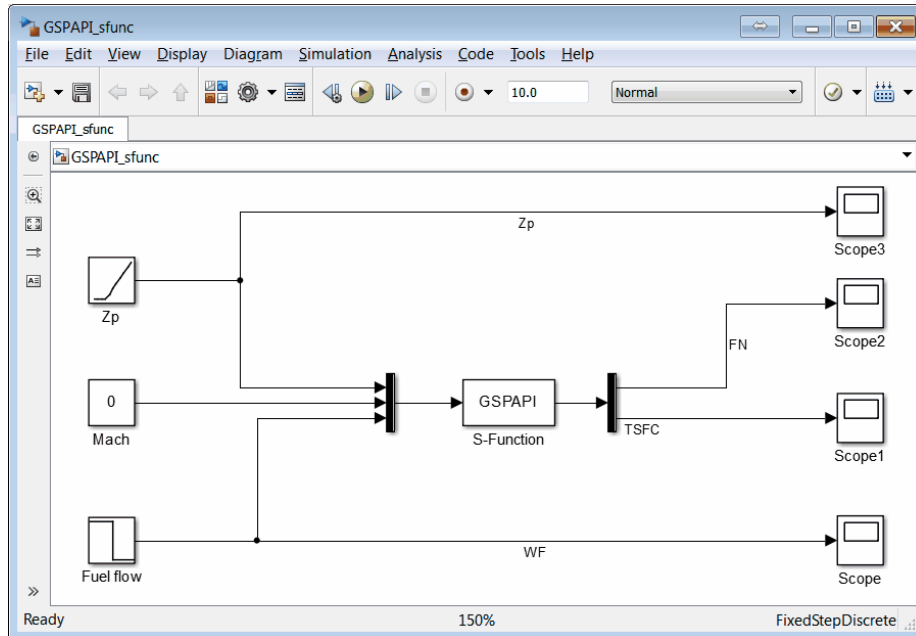
Compiling of the S-function is not required as the binary MATLAB MEX files are supplied. To set the input and output port width, the file `GSPAPI.cfg` has been supplied. This file is also used to set the model name of the GSP model that needs to be loaded in SIMULINK by the S-function. The configuration file (`GSPAPI.cfg`) should contain the following data:

```
# Configuration settings for GSPAPI  
INPUT_0_WIDTH=2  
OUTPUT_0_WIDTH=3  
GSPMDLFILENAME=YourModelFileName.mxl
```

Example:

1. Create a SIMULINK model `YourSimulinkModel.mdl` or `YourSimulinkModel.`

- slx in the directory described above
2. Create input signals and use a multiplexer to create a single input signal for a multi-port signal for the GSPAPI S-function
3. Insert a S-function and double click to change the S-function name to GSPAPI so that the GSPAPI.mexw32 DLL will be used
(do not forget to configure GSPAPI.cfg properly)
4. Place a de-multiplexer (demux) behind the S-function to split the multi-port signal in separate signals
5. Use "scope" components to visualize the signals



6. Run your simulation model



GSP

V

5 GSP ARP4868 API

Abbreviations

AIR	Aerospace Information Report
API	Application Programming Interface
ARP	Aerospace Recommended Practice
AS	Aerospace Standard
CEA	Chemical Equilibrium with Applications (NASA)
CFD	Computational Fluid Dynamics
DLL	Dynamic Link Library (also file extension)
EXE	Executable file (also file extension)
GSP	NLR's Gas turbine Simulation Program
GUI	Graphical User Interface
ISA	International Standard Atmosphere
NASA	National Aeronautics and Space Administration
NLR	National Aerospace Laboratory The Netherlands
OO	Object Orientation
OOP	Object-Oriented Programming
POD	Plain Old Data
SAE	Society of Automotive Engineers

Definitions and document conventions

CALLING PROGRAM / CALLER	The program responsible for interfacing with the ENGINE PROGRAM through the API functions defined in this document.
CUSTOMER/USER	The receiver and user of an ENGINE PROGRAM.
ENGINE PROGRAM	Gas turbine engine performance program. The API functions in this document are the interface to the usage of the engine program.
SUPPLIER	The supplier that created the ENGINE PROGRAM (NLR).

Conventions

Fixed width font (`Courier New`) is used to indicate source code (either Delphi or C/C++) or specific language reserved words to distinguish code from natural-language text. These fonts are so-called monospaced font types. Monospaced font types originate from typewriters where each character occupies the same width (horizontal space or fixed pitch). These fonts are still very popular amongst computer programmers as these fonts increase the readability of source code and are also used in terminal emulation as these fonts allow creating basic graphical lay outs for tabulated output for various tools (e.g. creating tables, list views, etc.)

```
// This is an example of the font in Courier New
```



5.1 Aerospace Recommended Practice 4868

The SAE S-15 Gas Turbine Performance Simulation Nomenclature and Interfaces committee has created a specification for standardizing communication between users (or customers) and developers (or suppliers) of gas turbine performance tools. The use of a standard interface not only simplifies the usage from the customer perspective (they can 'plug-in' different API software libraries without changing their caller application), but also simplifies the code development for the developer (input/output and other functionality has already been described how to communicate). The standard (recommended based on standard practice) is the ARP 4868. The ARP 4868 describes a set of functions and how they are expected to behave. These functions constitute a procedural or function based Application Program Interface (API) for gas turbine simulation programs. The customer using the API only needs to be aware of the different functions to be able to use the API. In other words when a gas turbine simulation environment is compliant with ARP 4868, the customers can acquire a copy of the ARP 4868 from SAE and implement the function calls inside its own application environment to allow gas turbine simulation calculations.

The ARP 4868 document defines generic language independent functions and specific appendices for implementations in programming language "C" and FORTRAN. The standard has no implementation examples for other programming languages yet (ARP 4868C is in development; this release will probably include more language implementations). Based on the implementation for the "C" programming language of the current ARP 4868 rev. B (released in May 2013), an implementation is written for the development of the API functions in Delphi (Embarcadero® Delphi® XE5 is NLR's current development tool for GSP).

The descriptions of input and output parameters nor the station numbering are covered by ARP 4868. The nomenclature of parameter naming and station numbering is covered by SAE AS755. A recommended practice has been defined that describes requirements for a gas turbine simulation environment developed under an Object Oriented (OO) modeling environment. This document has been prepared long after the development of GSP (GSP development started in 1996 and immediately implemented object orientation). As this recommended practice is based on the Numerical Propulsion System Simulation (NPSS) program written in C/C++, the document is biased to the implementation in NPSS. GSP has been developed with afore mentioned SAE standards in mind.

To use the GSP API functions of ARP 4868 it is required to purchase a copy of the ARP 4868 from SAE (website address <http://standards.sae.org/arp4868/>). Without this document, the API user has no knowledge of the available functions, what they do and how these should be called. The functions described in ARP 4868 and the accompanying text is subjected to copyright, this document therefore only describes additional documentation of input argument options for specific existing ARP 4868 functions (this is actually a requirement of the ARP 4868 standard) and documentation on custom developed functions that add extra functionality for the API user or the GSP API programmer.

5.1.1 ARP 4868 function conventions

All API functions of the engine program exist in the global name space having a unique name. A prefix for the function names is used for all function names in this API of the engine program to guarantee avoiding collision with other function names. The ARP 4868 document describes the API functions generically for all sorts of implementations of the interface; the implementation in this document describes the Delphi implementation of the API functions by means of a Dynamic-Link Library (DLL).

The lower case letter 'x' which is used as placeholders for implementations in different languages is be replaced to indicate the programming language the interface is written in. For the implementation in Delphi, the lowercase letter 'd' precedes the standard number to form the prefix 'd4868'. has been taken from ARP 4868, the 'd' has been added to obtain unique function names for an implementation of the ARP 4868 in the Delphi programming language; the function prototypes are available in ARP4868 Delphi Implementation.

Table 1 - Recommended 4868 function prefixes

Symbol	Language
c	C, C++
d	Delphi
f	FORTRAN
j	Java
pl	Perl
py	Python
vb	Visual Basic, VBA (Excel)

Function names are chosen carefully to represent the action the function performs in the engine program. These actions include: getting or setting data, perform program or log file initialization, parsing of additional input, run the model or terminate the model/engine program. Please note that different implementations of functions are present in the API when the function requires distinction between variable types. The data type for which the function is designed for, is represented in the function name by using a single character (supported data types are designated by "I" for integer, "F" for single precision floating point, "D" for double precision floating point, and "S" to represent a character string). Functions that do not operate on numerical data will not have these single character type indicators. Additionally a numerical value is added to the function name to indicate that array handling functions can return arrays of multiple dimensions.

5.1.2 Function implementation in GSP API

The table below describes the status of the implementation of the ARP 4868 functions in the GSP API. Note that an implementation status of marginal or partial means that the functions are implemented, but due to the difference of modelling environments, could not be fully complied. In those cases the ARP provides for marginal or partial implementation of the standard; i.e. only basic functionality of the function is achieved (partial implementation), or a warning is generated that the function is not supported (marginal implementation) (see [Error and warning messages](#) for a full list of the GSP API errors and warnings).

Table 2 – Function implementation matrix



		Implementation					
Function number	name	GSP API			ARP 4868 required		
		marginal	partial	full	full	full if supported	minimal
ARP4868.1	x4868activateLog			X			X
ARP4868.2	x4868closeLog			X			X
ARP4868.3	x4868defineDataList			X			X
ARP4868.4	x4868getArraySize[123]D	X				X	
ARP4868.5	x4868getDataList[IFD]			X			X
ARP4868.6	x4868getDataType		X				X
ARP4868.7	x4868getDescription	X					X
ARP4868.8	x4868getErrorMsg			X	X		
ARP4868.9	x4868get[IFDS]	X					X
ARP4868.10	x4868get[IFDS]1D	X					X
ARP4868.11	x4868get[IFDS][123]Dentry	X					X
ARP4868.12	x4868getSeverityMax	X					X
ARP4868.13	x4868getUnits			X			X
ARP4868.14	x4868initProg			X	X		
ARP4868.15	x4868isValidParamName			X	X		
ARP4868.16	x4868parseEfile	X					X
ARP4868.17	x4868parseFile	X					X
ARP4868.18	x4868parseString			X			X
ARP4868.19	x4868run			X	X		
ARP4868.20	x4868setDataList[IFD]	X					X
ARP4868.21	x4868set[IFDS]	X					X
ARP4868.22	x4868set[IFDS]1D	X					X
ARP4868.23	x4868set[IFDS][123]Dentry	X					X
ARP4868.24	x4868terminate			X	X		

The table above shows that all ARP 4868 required functions are implemented and all other functions at least have a marginally implementation to comply with the ARP4868B standard.

5.2 Function documentation

This chapter contains the necessary documentation to use the GSP API based on the ARP4868B standard implementation. The chapter describes additional documentation for using the standard ARP4868B functions, additional functions created for GSP API customers and additional functions created for GSP API developers.

5.2.1 Additional standard ARP4868B function documentation

This section describes additional documentation for using the functions with the GSP implementation of the ARP4868B standard. The ARP 4868 standard requires that optional argument settings are described for those functions that require mandatory or optional argument settings.

5.2.1.1 d4868initProg

Additional documentation to initialize the engine program

To initialize the engine program, the customer needs to load a model into the modeling simulation environment. The function takes 2 arguments (a list containing a series of strings (args) and the amount of strings (numArgs); see Table 3) which allow sending multiple options to the function. The current supported options are listed in Table 4; these options allow model loading and control of the model visibility and the visibility of the loading progress window. Note that an alternative for loading models is presented in section 3.2.2. When the

model is loaded using the `d4868initProg` function note that the '-fileName' option followed by the model filename in the next string value is then a required option (if model is loaded later with `d4868loadModel` from section 3.2.2, no required options are needed). Note to include path if the model is not in the folder of the caller application.

Table 3 - Arguments for `d4868initProg`

Parameter	In/Out	Description
numArgs	Input	Number of strings in the character array
args	Input	Character array containing initialization arguments
functStatus	Output	Function status of the <code>d4868initProg</code> function (function result)

The option '-createLog', followed by an optional string with a file name (if no file name specified, the output will be directed to the standard output channel: STDOUT), will create a log file during the initialization of the program. A separate call to create a log file after initialization will not be necessary then.

Table 4 - Additional argument values for `d4868initProgram` use with GSP API

Option/action	*args[i]	*args[i+1]	remark
Load a model by name	-fileName	string containing model (path and) name (required)	Required1)
Show model after loading	-showModel	string 'true' or 'false' (default) (optional argument value)	Optional2)
Show loading progress	-showProgress	string 'true' (default) or 'false' (optional argument value)	Optional2)
Create log file	-createLog	string containing log file (path and) name (optional)	Optional3)

1) In case function `d4868loadModel` from section 3.2.2 is used for separate loading of a model, no arguments are required for `d4868initProg`.

2) Only optional when the '-fileName' option is used.

3) Works similar as function `d4868activateLog` (log file will be created during `d4868initProg`, a separate call to `d4868activateLog` will not be necessary anymore)

5.2.1.2 d4868parseString

Additional documentation to parse a string

The rationale of this function is to have a function that allows the supplier of the API to define additional functions which are optional or required for usage of the specific implementation of the GSP engine program. The usage of this function is to pass a plain character string to the engine program (see Table 5). The engine program is responsible for parsing its contents, but the contents of the string must adhere to a predefined format described below. The string is parsed and any actions specified by the contents of the string are performed before this function returns. The string can be of any length and may contain spaces, special characters, statement termination characters implying any number of engine program statements contained within the specified string, etc.

The function returns the function result of the `d4868parseString` function (whether the function call succeeded or not); this does not reflect the return value of the operation resulting from executing the parsed string (when the result type of a parsed function is required, the function [d4868evalFunction\[IIDS\]](#) should be used).

Table 5 - Arguments overview for function `d4868parseString`

Parameter	In/Out	Description
-----------	--------	-------------

str	Input	String argument containing string to parse
functStatus	Output	Function status of the d4868parseString function (function result)

If a problem occurred during the parsing of the string, in the subsequent execution of the parsed contents or if the presented string is not supported, the function fails and will report an error. This function is implemented in the GSP API, however, its usage is limited as it is required to implement strings that can be parsed (see Table 6). Basically, the structure of a string that needs to be parsed is displayed graphically in Table 7. Note that word 1 and 2 are required, all other words are optional depending on the function that is called. An argument does not need to have a value, so if an argument is given, the value is optional depending on the function that is called. There is no limit on the amount of arguments other than if these are implemented.

Table 6 – Overview of supported (first) words for parsing

Action	Description
ExecFunction	String indicating that a function needs to be executed
ExecProcedure	String indicating that a procedure needs to be executed

Table 7 – Structure of the string to parse

Word	1	2	3	4	5	6
Description	Action ()	Func/ProcName	Argument 1	Value 1	Argument 2	Value 1
Example	ExecFunction	GSPEditComponent	Compressor01

The supported functions/procedures are published functions of the model. Through a special technique the string names and argument values can be parsed to load these model functions. There is no support for managing the return values of the functions with this function, the function [d4868evalFunction\[IFDS\]](#) should be used instead.

5.2.1.2.1 ExecFunction

When a string contains the keyword “ExecFunction” as first word in the string, the consecutive strings will be parsed to determine which function (the second word in the string) needs to be called. Arguments needed for the correct execution of the function should be placed after the second word. Table 8 shows an overview of the supported functions; the sections below will further explain the functions and show how to use them.

Table 8 – Overview of supported functions (ExecFunction)

Function	Description	Result
GSPEditComponent	Function to open a components data entry window	Boolean
GSPGetNrOfComponents	Returns the amount of model component blocks	Integer
GSPGetComponentNames	Returns a string with model component block names	String
GSPEditOutputOptions	Opens the model output options window	Boolean

5.2.1.2.1.1 GSPEditComponent

The function will try to execute the model function ‘GSPEditComponent’. This function opens the data entry input window of the component named in the third word. If a component named ‘Compressor01’ is present in the model, the function call will show the GUI data entry input window of the compressor with name ‘Compressor01’ allowing the customer to edit the model graphically. The string that needs to be parsed requires therefore 3 words.

The following code snippet shows usage of the `d4868parseString` function in Delphi.

```
aStringToParse := 'ExecFunction GSPEditComponent Compressor01';  
d4868parseString(PAnsiChar(aStringToParse));
```

This function returns a Boolean variable depending on the modal result status of closing the window. If the customer presses the 'OK' button the function returns 'TRUE', closing the window (top right red cross) or pressing the cancel button will return 'FALSE'. The `d4868parseString` has no support for managing the return values 'GSPEditComponent' returns. The result of the 'GSPEditComponent' function is usually not very important, if the result is required, the function [d4868evalFunction\[I/FDS\]](#) should be used instead.

5.2.1.2.1.2 GSPGetNrOfComponents

The function 'GSPGetNrOfComponents' is developed to return the amount of model components in the model. As `d4868parseString` has no support for managing the return values, this function has no use to be called by `d4868parseString`. Instead, this should be called by function [d4868evalFunction\[I/FDS\]](#). As that function internally calls the `d4868parseString` function, the function is explained here. The string that needs to be parsed requires 2 words. The result of the function 'GSPGetNrOfComponents' is an integer value containing the amount of model components in the model.

5.2.1.2.1.3 GSPGetComponentNames

The 'GSPGetComponentNames' function call returns a string containing all the model component names each separated by a line-break. As `d4868parseString` has no support for managing the return values, this function has no use to be called by `d4868parseString`. Instead, this should be called by function [d4868evalFunction\[I/FDS\]](#). As that function internally calls the `d4868parseString` function, the function is explained here. The string that needs to be parsed requires 2 words.

5.2.1.2.1.4 GSPEditOutputOptions

The function will execute the model function 'GSPEditOutputOptions'. This function opens the models output option data entry input window. The function returns true when the OK button has been pressed to exit the window (this is regarded that the input options have been changed), or false in any other case (the window is cancelled or aborted).

5.2.1.2.2 ExecProcedure

Although procedures are the perfect example of usage for the `d4868parseString` function (a procedure does not return a value), currently there are no implemented procedures.

5.2.1.3 d4868terminate

If a log file has been created but not manually closed with function `d4868closeLog`, the function `d4868terminate` will close the log automatically

5.2.2 Additional GSP specific functions

This section describes additional functions that are only available to the API users of GSP. These additional functions are implemented for simpler usage of the API, or as an extension of the current ARP4868B. Some of these functions could be proposed to be added to the ARP 4868 standard. Each function is described and function headers are supplied for Delphi and C/C++ generated caller programs. Note that these are exported functions available for the developers of caller programs.



5.2.2.1 d4868writeStrToLog

Write String to Log File

This function will write a string to the log file (or depending on the log file option, stdout). This function can be used by the customer to write any data string to the log file.

The additional functions are identified by a prefix of "a." followed by a function number similar to the ARP4868B document. Function "d4868writeStrToLog" is identified by function number "a.1". Below the prototypes for different implementations/programming languages is shown.

Delphi function prototype:

```
function d4868writeStrToLog(const logMessage: PAnsiChar): Integer;
```

C function prototype:

```
int c4868writeStrToLog(const char* logMessage)
```

5.2.2.2 d4868loadModel

Instructs Engine Program to Load a Model

The current ARP4868 implementation makes no distinction between the engine program and the engine model. This is because from legacy engine models, an engine program is considered to be a piece of code compiled into an executable. This results in a single model where the code and the input data are joined together. GSP on the contrary, is a generic gas turbine modeling simulation environment, loading the engine program does not automatically include loading a gas turbine model. A model contains the input data for a specific gas turbine configuration, or in case of GSP; a project containing multiple models and configurations which are hierarchically stored (using inheritance). From the stand-alone version of GSP you can load any model through load actions from the Graphical User Interface (GUI); this should be similar for a DLL simulation session. A function to load a model should therefore be required for GSP. Since no special builds of the GSP simulation program are made for a single engine model, the API should be equipped to load (and also close; see section 3.2.3) any gas turbine model prepared by the GSP stand-alone version.

Although this function explicitly loads a model, the documented function "d4868initProg" is also able to load a model using specific arguments. The reason for this separate model load function is that according to the ARP 4868 document, the function "d4868initProg" is only allowed to be run once. This latter constraint limits the engine program to load a single model. This constraint is lifted by the addition of separate functions to load and close (see section 3.2.3) gas turbine models.

This function could be suggested to be adopted into the ARP 4868 standard. However, every given engine program may require additional or different input, either structured input passed through one of the parse functions, or specific data values to be set before the model is loaded into the engine program. The program documentation should therefore specify all steps required to load the model into the engine program. The usage of this function will likely vary greatly between engine program environments; therefore unique documentation to use the function will need to be supplied. For usage in GSP, the "d4868loadModel" function must/may be called with additional options. These options consist of an identifier (prefixed by the minus '-' sign) and (optionally) a value that are contained in an array parameter which is communicated to the API together with the amount of strings the array consists of. The options for loading a GSP model using the API "d4868loadModel" function are listed in Table 9.

Table 9 - Parameter options for d4868loadModel function

Option/action	*args[i]	*args[i+1]	remark
---------------	----------	------------	--------

Load a model by name	-fileName	string containing model (path and) name	Required
Show model after loading	-showModel	string 'true' or 'false' (default)	Optional
Show loading progress	-showProgress	string 'true' (default) or 'false'	Optional

Note that the options are not case sensitive.

The “d4868loadModel” function is numbered “a.2”. The prototype for different implementations/programming languages is shown below.

Delphi function prototype:

```
function d4868loadModel(numArgs: Integer; args: PPAnsiChar):  
Integer;
```

C function prototype:

```
int c4868loadModel(int numArgs, char **args)
```

5.2.2.3 d4868closeModel

Instructs Engine Program to Close Model

The current ARP4868B implementation makes no distinction between the engine program and the engine model. GSP is a generic gas turbine modeling simulation environment; loading the simulation program does not automatically include loading a gas turbine model. In GSP, an engine simulation model is a specific combination of the modeling software environment (code) and an input file containing the data of the specific engine. A function to load a separate model is described in section 3.2.2. A function to close the model should be implemented as well.

Note that the standard function “d4868terminate” will close the current loaded model when called. The function “d4868closeModel” merely adds additional functionality when e.g. the customer requests that model should be changed during a simulation session.

This function could be suggested to be adopted into the ARP 4868 standard. However, every given engine program may require additional or different input, either structured input passed through one of the parse functions, or specific data values to be set before the model is closed from the engine program. The program documentation should therefore specify all steps required to close the model from the engine program. The usage of this function will likely vary greatly between engine program environments; therefore unique documentation to use the function will need to be supplied. For usage in GSP, the “d4868closeModel” function must/may be called with additional options. These options consist of an identifier (prefixed by the minus '-' sign) and (optionally) a value that are contained in an array parameter which is communicated to the API together with the amount of strings the array consists of. The options for closing a GSP model using the API “d4868closeModel” function are listed in Table 10.

The “d4868closeModel” function is numbered “a.3”. The prototype for different implementations/programming languages is shown below.

Delphi function prototype:

```
function d4868closeModel(numArgs: Integer; args: PPAnsiChar):  
Integer;
```

C function prototype:

```
int c4868closeModel(int numArgs, char **args)
```

Table 10 - Parameter options for d4868closeModel function

Option/action	*args[i]	*args[i+1]	remark
---------------	----------	------------	--------



Show save dialog	-saveDlg	string: 'true' or 'false' (default)	Optional
------------------	----------	-------------------------------------	----------

Note that the options are not case sensitive.

5.2.2.4 d4868saveModel

Instructs Engine Program to Save Model

If a loaded model is deliberately changed by the customer, an option to save the model should be provided. The function for this purpose is numbered “a.4” and called “d4868saveModel”. The function could be suggested for implementation in the ARP 4868 standard. The usage of this function will then likely vary greatly between engine program environments; therefore unique documentation to use this function needs to be supplied. These options consist of an identifier (prefixed by the minus '-' sign) and (optionally) a value that are contained in an array parameter which is communicated to the API together with the amount of strings the array consists of. The options for saving a GSP model using the API “d4868saveModel” function are listed in Table 11.

Prototypes for function “d4868saveModel” with number “a.4”:

Delphi function prototype:

```
function d4868saveModel(numArgs: Integer; args: PPAnsiChar):  
Integer;
```

C function prototype:

```
int c4868saveModel(int numArgs, char **args)
```

Note that the function will save the current model under the current name if no arguments are given. To save the model under a different name, the implementation of d4868saveModel accepts the following arguments for parameter args:

Table 11 - Parameter options for d4868saveModel function

Action	*args[i]	*args[i+1]	remark
Save model as	-saveAs	string containing alternative model (path and) name	Optional

Note that the options are not case sensitive.

5.2.2.5 d4868showModel

Instructs Engine Program to Show Model

If a model is loaded into the engine program, the customer may want to interact with the model. The function to show/hide a model is called “d4868showModel” and is numbered by “a.5”. By showing the model, the model window will be shown on the screen, and will be available for user interfacing.

The usage of this function may be subjected to change (e.g. allowing editing of the model). Allowing more options in the future implies to have a variable length of parameters or function arguments. This is implemented by passing an amount of array parameters and a list of strings (options consist of an identifier (prefixed by the minus '-' sign) and (optionally) a value that are contained in the string array) The options for showing a GSP model using the API “d4868showModel” function are listed in Table 12.

Prototypes for function “d4868showModel” with number “a.5”:

Delphi function prototype:

```
function d4868showModel(numArgs: Integer; args: PPAnsiChar):  
Integer;
```

C function prototype:


```
int c4868showModel(int numArgs, char **args)
```

Note that the function will show/hide the current model if no arguments are given by reversing the current value of the models visibility value (when the model is shown a call to `d4868showModel(0, nil)` in Delphi or `d4868showModel(0, null)` in 'C' will hide the model and vice versa). The implementation of `d4868showModel` accepts the arguments listed in Table 12 for parameter args.

Table 12 - Parameter options for `d4868showModel` function

Action	*args[i]	*args[i+1]	remark
Show model	-show	string containing the string 'TRUE' or 'FALSE'	Optional

Note that the options are not case sensitive.

5.2.2.6 d4868getInputDataList

Get a List of Input Parameters to Set Later

Due to the different approaches of the development of gas turbine simulation software, GSP does not allow to just set any arbitrary parameter in the engine model. GSP uses a strict model configuration and case management system to ensure the model is properly configured. The roots for this are firmly embedded in the chosen GUI design and approach. A strict distinction is made between user input variables and output variables/parameters. This function will return the available input parameters (component name and input parameter name separated by a dot '.'; note that space characters are purged from the component name) that can be set in the loaded model, a list ID will be assigned to the list that is created by running this function (the input parameter data list always has ID value 999 by design, user created parameter lists start with 1000). It should be noted that this list is set by the creator of the GSP model and can only be altered or extended with a stand-alone version of GSP (which is supplied with the GSP API installer package). It is not allowed to run this function multiple times as it serves no purpose as the input parameter list is fixed by the creator of the engine model. The function will return a list of input parameter names, a datatype and an ID. The function requires knowing beforehand how many parameters are to be expected. This is required prior to calling the function as this function is concerned with proper management of the allocation of memory. The function that retrieves the correct amount of input parameters is called "[d4868getInputDataListSize](#)". An overview of the input/output parameters is given in Table 13.

Prototypes for function "d4868getInputDataList" with number "a.6":

Delphi function prototype:

```
function d4868getInputDataList(paramList: PParamList;
                               size: Integer;
                               dataType: PParamType;
                               listID: PInteger): Integer;
```

C function prototype:

```
int c4868getInputDataList (char **paramList, int size,
                           char *dataType , int *listID)
```

Table 13 – Description of arguments for `d4868getInputDataList` function

Argument	Type	In/Output	Remark
paramList	pointer to character pointer	output	List of parameter names, must all be scalar and of the same type
size	integer	input	Number of names in the list
dataType	pointer to character	output	Calling program data type for all Engine



e			program paramList variables
listID	pointer to integer	output	Integer identifier associated with this defined list of names

The function should at least give a warning when the input data list is larger than the reserved memory. At all times, the `paramList` cannot contain more items than defined by `size`.

5.2.2.7 d4868getInputDataListSize

Get List Size of Input Parameters List

This function retrieves the amount of specified input parameters and the length of the largest parameter name. To correctly store the input parameter list, the caller program needs to allocate the proper amount of memory (this is the responsibility of the caller as any memory allocated by the engine program will get out of scope when the function call ends). The function takes 2 arguments, which are references to variables allocated by the caller application, to store the resulting amount of input parameters and the longest string size (see Table 14).

Prototypes for function "d4868getInputDataListSize" with number "a.7":

Delphi function prototype:

```
function d4868getInputDataListSize(listSize, maxSizeString:  
PInteger): Integer;
```

C function prototype:

```
int d4868getInputDataListSize (int *size, int *maxSizeString)
```

Table 14 – Description of arguments for d4868getInputDataListSize function

Argument	Type	In/Output	Remark
listSize	pointer to integer	output	Number of names in the list
maxSizeString	pointer to integer	output	Maximum string length

5.2.2.8 d4868programInfo

Get Program Information

The "d4868programInfo" function provides overall information about the engine program. This includes strings representing the engine program name and an engine program version or identifier number. The engine program version argument contains non-numeric characters to separate main version from minor, release and build version (separated by the dot character).

Note that this is not a function to retrieve the status of the model, a separate equation should be created for that purpose (such a function could be added in the next release of the GSP API).

A similar function existed in the previous implementation of the ARP4868 (this is the "rev A" or initial version). However, this implementation is more minimalistic and does not return e.g. model status.

Prototypes for function "d4868programInfo" with number "a.8":

Delphi function prototype:

```
function d4868programInfo(progName, progVersion: PAnsiChar):
```

Integer;

C function prototype:

```
int d4868programInfo (char *progName, char *progVersion)
```

Table 15 – Description of arguments for d4868programInfo function

Argument	Type	In/Output	Remark
progName	pointer to character pointer	output	Program name string
progVersion	pointer to character pointer	output	Version number string

5.2.2.9 d4868evalFunction[IFDS]

Get a scalar return value from a function call

The current ARP4868B standard allows for executing functions from a string argument. The current implementation lacks a proper implementation of calling a function and retrieving the return result. Function d4868evalFunction[IFDS] evaluates a function and returns the result of a function (e.g. if the calling of the function succeeded or not). A plain character string argument is used to provide the function call information. The string can be of any length, it just needs to reference an existing engine program function utilizing the proper call sequence. Table 16 shows the description of the function arguments.

Prototypes for function "d4868evalFunction[IFDS]" with number "a.9":

Delphi function prototype:

```
function d4868evalFunctionI(const paramName: PAnsiChar; value:
PInteger): Integer;
function d4868evalFunctionF(const paramName: PAnsiChar; value:
PFloat): Integer;
function d4868evalFunctionD(const paramName: PAnsiChar; value:
PDouble): Integer;
function d4868evalFunctionS (const paramName: PAnsiChar; value:
PAnsiChar; buffSize: Integer): Integer;
```

C function prototype:

```
int d4868evalFunctionI(char *paramName, int *value)
int d4868evalFunctionF(char *paramName, float *value)
int d4868evalFunctionD(char *paramName, double *value)
int d4868evalFunctionS(char *paramName, char *value, int buffSize)
```

Common error conditions:

- The input string is not referencing an existing function or the usage is incorrect.
- Function was called before the x4868initProg function.
- The Engine program does not support the parsing of strings.

A minimal implementation of this function that generates an error when called is acceptable.

Table 16 – Description of arguments for d4868getInputDataListSize function

Argument	Type	In/Output	Remark
paramName	character pointer (string)	input	Function call to parse
value	Pointer to integer, float or double	output	The parsed 'paramName' function result

buffSize	integer	input	Size of the reserved memory
----------	---------	-------	-----------------------------

5.2.3 Functions for the GSP API developer

This section describes functions that have been developed and added as additional helper functions to the API using the ARP 4868 standard. These functions have been added to simplify code development and maintenance by reducing repeating code constructions by the development of functions. Note that these functions are **not exported** and as such not available for the developers of caller programs. These are merely added to complete the documentation of the API development to aid GSP partner API development.

5.2.3.1 d4868initProgInitialized

Engine program initialized?

This function is used to determine if the calling of this API function is preceded by the initialization of the engine program (calling the engine program initialization function, `d4868initProg`, number 14 from ARP4868B). Depending on the result of the `d4868initProgInitialized` function, the programmer decides to either continue function execution or exit the function. All functions declared by the ARP4868B standard (except for the `d4868initProg` function) require that the engine program is initialized before calling any other function.

Note that these functions are specifically written for the GSP API development in Delphi; hence no prototypes for the C/C++ language are given. The Delphi function prototype is:
function `d4868initProgInitialized`: **Boolean**;

The function will return a Boolean parameter based on whether the ARP4868B `d4868initProg` has been executed prior to the calling of the function this `d4868initProgInitialized` function is used in. This function is typically one of the first functions that should be called when developing API functions.

Typical usage of the `d4868initProgInitialized` function is to check the result and decide to continue or exit the function in which `d4868initProgInitialized` has been called. Before exit, setting an appropriate error value is strongly advised. The following code snippet shows the typical usage of the `d4868initProgInitialized` function.

```
// Default exit if "d4868initProg" has not been called prior to this
function
if not d4868initProgInitialized then
begin
    // Some error: Function "XXX" called prior to function "d4868initProg"!
    d4868setFunctionError(.....);
    Exit;
end;
```

5.2.3.2 d4868getAPIInterfaceComponent

Get API Interface Model Component

This function returns the API interface component of the GSP model. This component is an essential model component as it provides the interface for the input variables. Without such a component, you cannot run GSP models with the GSP API.

The Delphi function prototype is:

```
function d4868getAPIInterfaceComponent: TAPIInterfaceCompForm;
```

When the `d4868getAPIInterfaceComponent` function is called, the function returns the API interface object reference or nil (the nil constant is a pointer value that is defined as undetermined, an equivalent in C-code exists: null). With this reference, the programmer is able to access the interfacing component model which contains input and output parameters. When the result is nil, the interfacing component is not found, meaning that the model is incorrectly configured to act as an API engine model.

5.2.3.3 d4868initFunction

Initialize the Function to Reset Error code and Message

This function initializes the global variables that contain the information of the last known error code and error message. Passing the `Result` parameter of the calling function (which will be passed on in `(FunctionResultVar)` to the `d4868initFunction` will reset the function results parameter to 0 (zero). This function is typically the first functions that should be called when developing API functions.

5.2.3.4 d4868setFunctionError

Set Error Code and Message

This function is used to finalize a function and should be called prior to exiting a function when the function encountered an error. Alternatively this function may be called when the function produces a warning message (no exit of the function that calls this function in case of a warning).

The Delphi function prototype is:

```
procedure d4868setFunctionError(var FunctionResultVar: Integer;  
                               ResultCode: Integer;  
                               ResultMessage: AnsiString);
```

The function has 2 input parameters, the `ResultCode` and the `ResultMessage`, and 1 output parameter, `FunctionResultVar` (the `Result` parameter of the calling function should be passed here).

5.2.3.5 d4868modelLoaded

Is a Model Loaded?

This function is used to determine if a model has been loaded into memory.

The Delphi function prototype is:

```
function d4868modelLoaded(ShowMessage,  
                           ShowNotLoadedError: Boolean):Boolean;
```

The function will return a `Boolean` parameter based on whether a model has been loaded (`result` is `True`) or not (`result` is `False`). The function takes 2 input arguments. The first argument, `ShowMessage`, specifies whether a message should be displayed at all. The second argument, `ShowNotLoadedError`, specifies whether a message should be displayed only when no model is loaded. For API usage the most common call to this function is e.g.:

```
IsModelLoaded := d4868modelLoaded(False, False);
```



or

```
if not d4868modelLoaded(False, False) then  
  
begin  
  
...
```

5.2.3.6 d4868GetIndexOfOption

This function returns the integer index of the specified option string in a given string list as the function results value. If not present in the list, the function returns the integer '-1'. The function expects 3 input arguments that pass the option name to be found in the string list, the number of strings in the string list and the string list itself. This list of strings contains the specified amount of options and optional option values. This implies that the *i*th string in the list is the option (prefixed by a 'dash'), and the (*i*+1)th string in the list is the optional value of that option.

The Delphi function prototype is:

```
function d4868GetIndexOfOption(OptionString: AnsiString;  
                               numArgs: Integer; args: PPAnsiChar)  
                               : Integer;
```

The function will return the index of the value of the option in the argument list

5.2.3.7 d4868GetValueOfOption

Function that returns the value string of the option in an argument list. This function is quite similar to the previous function except for the difference that not the index, but the string value is returned. This implies that for a given option (to find if it exists the programmer needs to call d4868GetIndexOfOption and test if its position is > -1) the string value of the next entry in the array is returned (if a value is present, else an empty string is returned).

The Delphi function prototype is:

```
function d4868GetValueOfOption(OptionString: AnsiString;  
                               numArgs: Integer; args: PPAnsiChar)  
                               : AnsiString;
```

5.2.3.8 d4868listToString

This function creates a string from an array of strings (so called list). This string can then be used for e.g. printing to log files or to the screen.

The Delphi function prototype is:

```
function d4868listToString(numArgs: Integer; args: PPAnsiChar;  
                           StartString, SeparatorString, EndString  
                           : AnsiString): AnsiString;
```

Provide the amount of array arguments, the array, a starting string, a separator string and an end string to customize the output string. For API usage the most common call to this function is e.g.:

```
d4868listToString(numArgs, args, '[' , ',' , ']').
```

This will create a comma (and space) separated string of a certain amount of array parameters enclosed by square brackets.

5.2.3.9 d4868FreeAll

This is an internal function that has restricted use for the GSP API development team as this function will free reserved memory that has to be released prior to exiting the DLL. This is typically one of the last functions that need to be called prior to closing the engine program environment.

The Delphi function prototype is:

```
function d4868FreeAll: Boolean;
```

The function will return a Boolean parameter based on whether the freeing of all reserved memory has succeeded.

5.3 Using the GSP ARP4868 API

GSP API model configuration

To apply the GSP API to a custom off-line simulation run from a different program other than the GSP stand-alone application, another program or programming language should be used to load and run the API and its associated functions. Before running a model, a model needs to be created or set-up with the standard stand-alone GSP application. Basically any existing GSP model can be used by the GSP API. Configure your model as you would run it in the stand-alone version: configure the cycle design and sizing data in so-called configuration nodes and when finished instantiate a case model. To run a model with the GSP API the model requires an API component block on the case model. This specific API model component is used to set the case input data (the input parameters that control the model input for the simulation). This component can be dragged from the GSP API component library (see Figure 4) and placed onto the (case) model work sheet (see Figure 5).



Figure 4 - GSP API component library

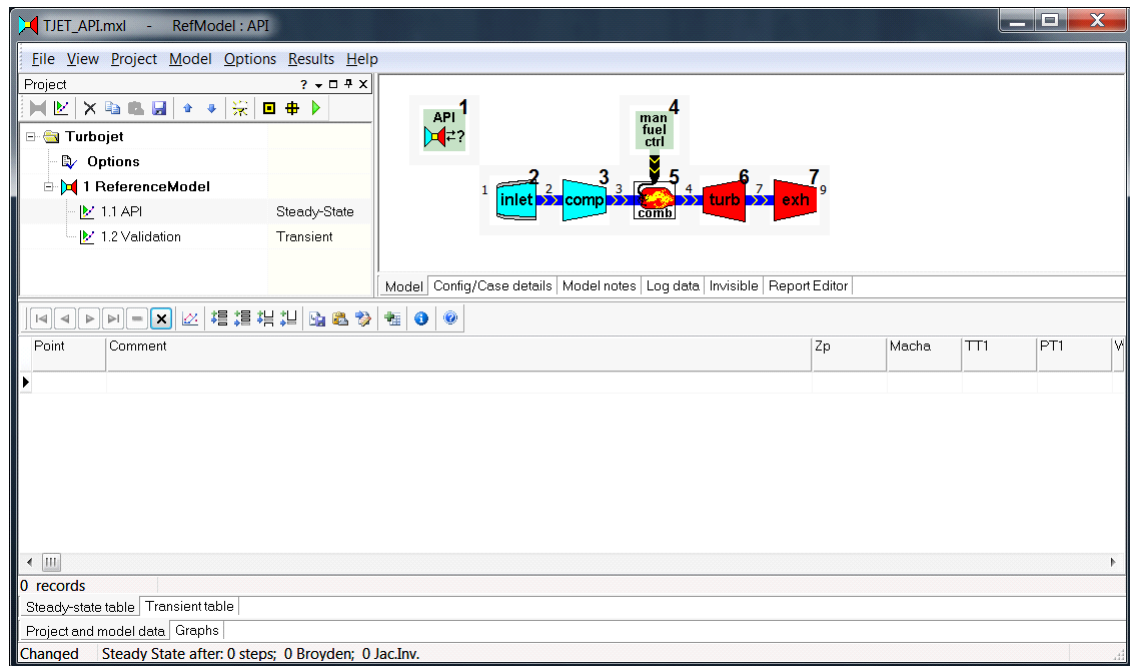


Figure 5 - GSP API model

GSP API model component configuration

To provide the GSP API user input parameters to control the model, the model needs to be correctly configured. The GSP API model component data entry window provides the necessary interfacing elements to setup input parameters for the GSP API model user. The "Input" tab sheet (see Figure 6) provides the selection of (available) input parameters as determined by the model configuration. One can only control the fuel flow value when the "Wf" has been selected as control parameter; if the modeler requires to control the cycle using the combustor exit temperature, the manual fuel controller control parameter should be set to "Texit"; i.e. on the "General" tab sheet of the data entry window (see Figure 7).

When the list of input parameters is selected, the check mark of the column entitled "Active" determines whether the parameter can be used with the GSP API. Save the model when the model is equipped with a list of input parameters; it is now ready to be used by the API. Note that the model is configured in Steady State case mode. This run-case mode requires the user to specify the input for each run. The next chapter discusses 2 console applications that use a model that has been setup according to this chapter.

API interface window

API case control1 ID string api Units As Model Calc.Nr. 1

Input Output Options Remarks

Navigation buttons: < << >> > A + - X

Index	Model Component	Input parameter	Value	Unit	Active
1	Ambient Conditions	Zp	0	[m]	<input checked="" type="checkbox"/>
2	Ambient Conditions	Mach	0	[-]	<input checked="" type="checkbox"/>
3	Manual Fuel Control	Wf	0.38	[kg/s]	<input checked="" type="checkbox"/>

OK Cancel Help

Figure 6 - GSP API component model data entry window

API interface window

API case control1 ID string api Units As Model

Input Output Options Remarks

Navigation buttons: < << >> > A + - X

Index	Model Component	Input parameter	Value
1	Ambient Conditions	Zp	
2	Ambient Conditions	Mach	
3	Manual Fuel Control	Humidity Mass%	

Dropdown menu for Index 2: Mach, Humidity Mass%, Mach, Zp

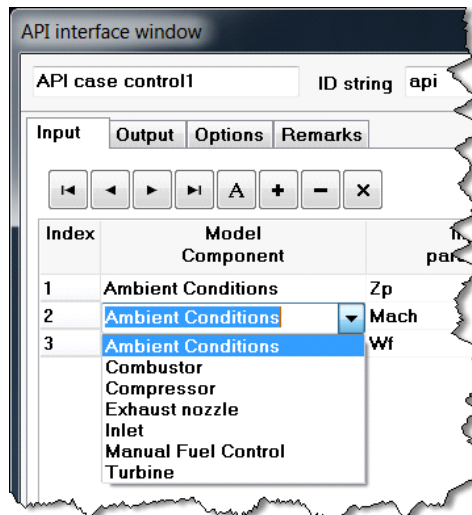
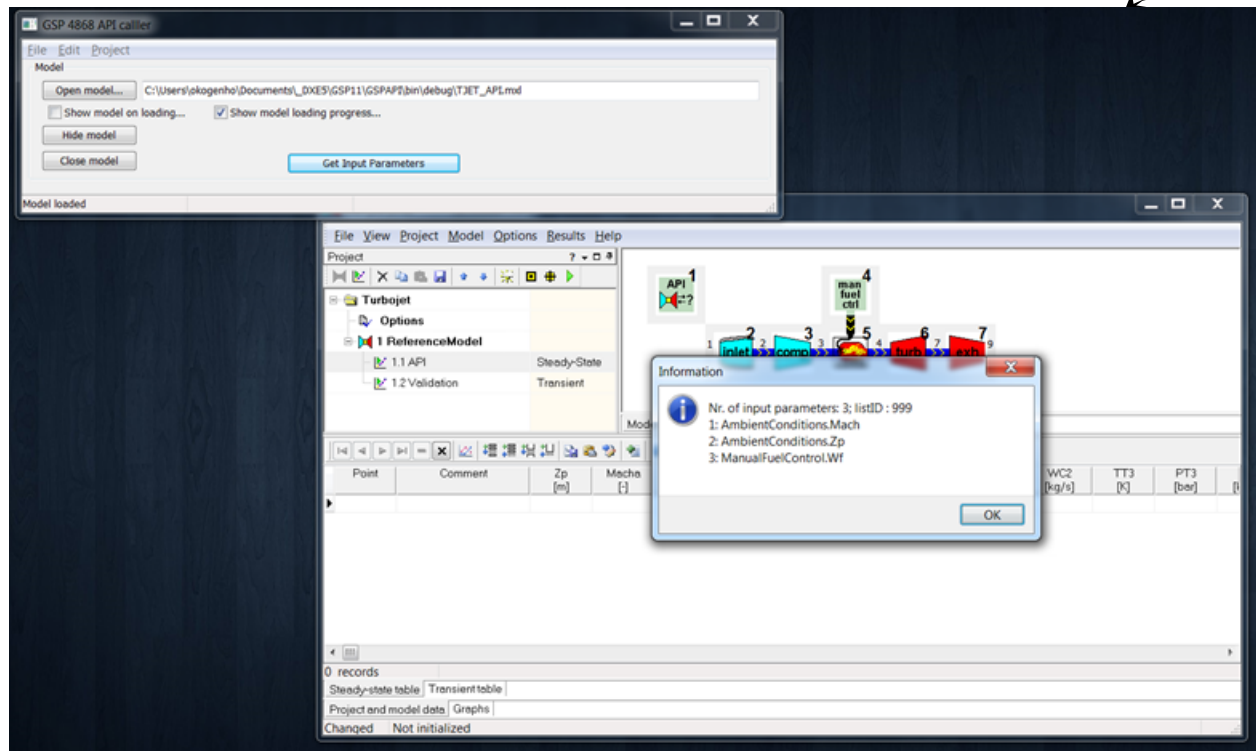


Figure 7 - GSP API component model data entry window; setting input parameters (left: model component selection, right: input parameter selection)

5.3.1 Examples

The development of a caller application is required to implement the GSP API in custom applications. The GSP API is delivered as a single DLL file that can be used on the Microsoft® Windows® platform. This DLL file needs to be loaded into memory and in order to use the functions to simulate a gas turbine engine, the functions need to be found in the file and address pointers to these entry points stored. The next sections demonstrate the usage of the GSP API for two different programming languages. Note that because the GSP API has been developed using standard parameter types, the same DLL can be used in Delphi and in C/C++ created programs.

This section shows some examples to use the GSP API ARP 4868 functions. An example console application in Delphi is shown in [Console application in Delphi](#) and an example console application in C++ is shown in [Console application in C/C++](#). Note that a caller application is not limited to being a console type application; a GUI based application could be developed instead. Such an application would be interesting to be used as a performance data deck generator where the customer will be able to define the input data (e.g. power setting, power off-take and flight conditions) using a GUI. A screenshot of a very basic API GUI application is shown below.



5.3.1.1 Console application in C/C++

Application source code

```
#pragma hdrstop
#pragma argsused

#include <stdio.h>
#include <tchar.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <windows.h>
#include <string.h>

#define DLLFILENAME "GSP.dll" // Name of the dll file
#define GSPMDLFILENAME "TJET_API.mxl"

// ARP 4868 functions
typedef int (*_d4868activateLog) (char *);
typedef int (*_d4868closeLog) (void);
typedef int (*_d4868defineDataList) (char **, int, char *, int *);
//typedef int (*_d4868getArraySize1D) (char *, int *);
//typedef int (*_d4868getArraySize2D) (char *, int *, int *);
//typedef int (*_d4868getArraySize3D) (char *, int *, int *, int *);
typedef int (*_d4868getDataListD) (int, int, double *);
typedef int (*_d4868getDataListF) (int, int, float *);
typedef int (*_d4868getDataListI) (int, int, int *);
//typedef int (*_d4868getDataType) (char *, char *, int);
//typedef int (*_d4868getDescription) (char *, char *, int);
typedef int (*_d4868getErrMsg) (char *, int);
//typedef int (*_d4868getI) (char *, int *);
//typedef int (*_d4868getF) (char *, float *);
//typedef int (*_d4868getD) (char *, double *);
//typedef int (*_d4868getS) (char *, char *, int);
//typedef int (*_d4868getI1D) (char *, int *, int);
//typedef int (*_d4868getF1D) (char *, float *, int);
//typedef int (*_d4868getD1D) (char *, double *, int);
//typedef int (*_d4868getI1Dentry) (char *, int, int *);
```



```
//typedef int (*_d4868getI2Dentry) (char *, int, int, int *);
//typedef int (*_d4868getI3Dentry) (char *, int, int, int, int *);
//typedef int (*_d4868getF1Dentry) (char *, int, float *);
//typedef int (*_d4868getF2Dentry) (char *, int, int, float *);
//typedef int (*_d4868getF3Dentry) (char *, int, int, int, float *);
//typedef int (*_d4868getD1Dentry) (char *, int, double *);
//typedef int (*_d4868getD2Dentry) (char *, int, int, double *);
//typedef int (*_d4868getD3Dentry) (char *, int, int, int, double *);
//typedef int (*_d4868getS1Dentry) (char *, int, char *, int);
//typedef int (*_d4868getS2Dentry) (char *, int, int, char *, int);
//typedef int (*_d4868getSeverityMax) (int *);
typedef int (*_d4868getUnits) (char *, char *, int);
typedef int (*_d4868initProg) (int, char **);
typedef int (*_d4868isValidParamName) (char *, int *);
//typedef int (*_d4868parseEfile) (char *);
//typedef int (*_d4868parseFile) (char *);
//typedef int (*_d4868parseString) (char *);
typedef int (*_d4868run) (void);
//typedef int (*_d4868setDataListD) (int, int, double *);
//typedef int (*_d4868setDataListF) (int, int, float *);
//typedef int (*_d4868setDataListI) (int, int, int *);
//typedef int (*_d4868setI) (char *, int);
//typedef int (*_d4868setF) (char *, float);
//typedef int (*_d4868setD) (char *, double);
//typedef int (*_d4868setS) (char *, char *);
//typedef int (*_d4868setI1D) (char *, int *, int);
//typedef int (*_d4868setF1D) (char *, float *, int);
//typedef int (*_d4868setD1D) (char *, double *, int);
//typedef int (*_d4868setI1Dentry) (char *, int, int);
//typedef int (*_d4868setI2Dentry) (char *, int, int, int);
//typedef int (*_d4868setI3Dentry) (char *, int, int, int, int);
//typedef int (*_d4868setF1Dentry) (char *, int, float);
//typedef int (*_d4868setF2Dentry) (char *, int, int, float);
//typedef int (*_d4868setF3Dentry) (char *, int, int, int, float);
//typedef int (*_d4868setD1Dentry) (char *, int, double);
//typedef int (*_d4868setD2Dentry) (char *, int, int, double);
//typedef int (*_d4868setD3Dentry) (char *, int, int, int, double);
//typedef int (*_d4868setS1Dentry) (char *, int, char *);
//typedef int (*_d4868setS2Dentry) (char *, int, int, char *);
typedef int (*_d4868terminate) (void);

// Additional non-ARP 4868 functions
typedef int (*_d4868writeStrToLog) (char *);
//typedef int (*_d4868loadModel) (int, char **);
//typedef int (*_d4868closeModel) (int, char **);
//typedef int (*_d4868saveModel) (int, char **);
//typedef int (*_d4868showModel) (int, char **);
//typedef int (*_d4868getInputDataList) (char **, int, char *, int *);
//typedef int (*_d4868getInputDataListSize) (int *);
//typedef int (*_d4868programInfo) (char *, char *);
//typedef int (*_d4868evalFunctionI) (char *, int *);
//typedef int (*_d4868evalFunctionF) (char *, float *);
//typedef int (*_d4868evalFunctionD) (char *, double *);
//typedef int (*_d4868evalFunctionS) (char *, char *, int);

char *dllname = DLLFILENAME; // GSP DLL name in local directory
static HINSTANCE hInstanceControl;

// ARP 4868 functions
_d4868activateLog d4868activateLog = NULL;
_d4868closeLog d4868closeLog = NULL;
_d4868defineDataList d4868defineDataList = NULL;
//_d4868getArraySize1D d4868getArraySize1D = NULL;
//_d4868getArraySize2D d4868getArraySize2D = NULL;
//_d4868getArraySize3D d4868getArraySize3D = NULL;
_d4868getDataListD d4868getDataListD = NULL;
_d4868getDataListF d4868getDataListF = NULL;
_d4868getDataListI d4868getDataListI = NULL;
//_d4868getDataType d4868getDataType = NULL;
//_d4868getDescription d4868getDescription = NULL;
_d4868getErrMsg d4868getErrMsg = NULL;
//_d4868getI d4868getI = NULL;
//_d4868getF d4868getF = NULL;
//_d4868getD d4868getD = NULL;
```



```

//_d4868getS d4868getS = NULL;
//_d4868getI1D d4868getI1D = NULL;
//_d4868getF1D d4868getF1D = NULL;
//_d4868getD1D d4868getD1D = NULL;
//_d4868getI1Dentry d4868getI1Dentry = NULL;
//_d4868getI2Dentry d4868getI2Dentry = NULL;
//_d4868getI3Dentry d4868getI3Dentry = NULL;
//_d4868getF1Dentry d4868getF1Dentry = NULL;
//_d4868getF2Dentry d4868getF2Dentry = NULL;
//_d4868getF3Dentry d4868getF3Dentry = NULL;
//_d4868getD1Dentry d4868getD1Dentry = NULL;
//_d4868getD2Dentry d4868getD2Dentry = NULL;
//_d4868getD3Dentry d4868getD3Dentry = NULL;
//_d4868getS1Dentry d4868getS1Dentry = NULL;
//_d4868getS2Dentry d4868getS2Dentry = NULL;
//_d4868getSeverityMax d4868getSeverityMax = NULL;
_d4868getUnits d4868getUnits = NULL;
_d4868initProg d4868initProg = NULL;
_d4868IsValidParamName d4868IsValidParamName = NULL;
//_d4868parseEfile d4868parseEfile = NULL;
//_d4868parseFile d4868parseFile = NULL;
//_d4868parseString d4868parseString = NULL;
_d4868run d4868run = NULL;
//_d4868setDataListD d4868setDataListD = NULL;
//_d4868setDataListF d4868setDataListF = NULL;
//_d4868setDataListI d4868setDataListI = NULL;
//_d4868setI d4868setI = NULL;
//_d4868setF d4868setF = NULL;
//_d4868setD d4868setD = NULL;
//_d4868setS d4868setS = NULL;
//_d4868setI1D d4868setI1D = NULL;
//_d4868setF1D d4868setF1D = NULL;
//_d4868setD1D d4868setD1D = NULL;
//_d4868setI1Dentry d4868setI1Dentry = NULL;
//_d4868setI2Dentry d4868setI2Dentry = NULL;
//_d4868setI3Dentry d4868setI3Dentry = NULL;
//_d4868setF1Dentry d4868setF1Dentry = NULL;
//_d4868setF2Dentry d4868setF2Dentry = NULL;
//_d4868setF3Dentry d4868setF3Dentry = NULL;
//_d4868setD1Dentry d4868setD1Dentry = NULL;
//_d4868setD2Dentry d4868setD2Dentry = NULL;
//_d4868setD3Dentry d4868setD3Dentry = NULL;
//_d4868setS1Dentry d4868setS1Dentry = NULL;
//_d4868setS2Dentry d4868setS2Dentry = NULL;
_d4868terminate d4868terminate = NULL;

// Additional non-ARP 4868 functions
_d4868writeStrToLog d4868writeStrToLog = NULL;
//_d4868loadModel d4868loadModel = NULL;
//_d4868closeModel d4868closeModel = NULL;
//_d4868saveModel d4868saveModel = NULL;
//_d4868showModel d4868showModel = NULL;
//_d4868getInputDataList d4868getInputDataList = NULL;
//_d4868getInputDataListSize d4868getInputDataListSize = NULL;
//_d4868programInfo d4868programInfo = NULL;
//_d4868evalFunctionI d4868evalFunctionI = NULL;
//_d4868evalFunctionF d4868evalFunctionF = NULL;
//_d4868evalFunctionD d4868evalFunctionD = NULL;
//_d4868evalFunctionS d4868evalFunctionS = NULL;

int _tmain(int argc, _TCHAR* argv[])
{
    char **ArrayOfStrings;
    char *ParamList[] = {"TT1", "TT2", "TT3", "TT4", "TT5", "TT9"};
    char *ParamList1[] = {"PT1", "PT2", "PT3", "PT4", "PT5", "PT9"};
    char *ParamList2[] = {"W1", "W2", "W3", "W4", "TSFC", "FN"};
    char **PParamList;
    char *DataType;
    char DataTypeString[20], UnitString[20];
    int ListID, ListID1, ListID2, i;
    int IsValid, variableNumberOfElements;
    char ErrorMessage[256];

```



```
double *DArrayData;
float *FArrayData;
int *IArrayData;

/* Need to get information to load the DLL that helps link to the model code */
hInstanceControl = LoadLibrary(dllname);
// If the DLL was found and loaded, we can now get the routine addresses for calling.
// This will get all the functions to be available for later use.

if( hInstanceControl != NULL){
    // ARP 4868 functions
    d4868activateLog = (_d4868activateLog)GetProcAddress(hInstanceControl, "d4868activateLog");
    d4868closeLog = (_d4868closeLog)GetProcAddress(hInstanceControl, "d4868closeLog");
    d4868defineDataList = (_d4868defineDataList)GetProcAddress(hInstanceControl,
        "d4868defineDataList");
    // d4868getArraySize1D = (_d4868getArraySize1D)GetProcAddress(hInstanceControl,
    //     "d4868getArraySize1D");
    // d4868getArraySize2D = (_d4868getArraySize2D)GetProcAddress(hInstanceControl,
    //     "d4868getArraySize2D");
    // d4868getArraySize3D = (_d4868getArraySize3D)GetProcAddress(hInstanceControl,
    //     "d4868getArraySize3D");
    d4868getDataListD = (_d4868getDataListD)GetProcAddress(hInstanceControl, "d4868getDataListD"
);
    d4868getDataListF = (_d4868getDataListF)GetProcAddress(hInstanceControl, "d4868getDataListF"
);
    d4868getDataListI = (_d4868getDataListI)GetProcAddress(hInstanceControl, "d4868getDataListI"
);
    // d4868getDataType = (_d4868getDataType)GetProcAddress(hInstanceControl,
    "d4868getDataType");
    // d4868getDescription = (_d4868getDescription)GetProcAddress(hInstanceControl,
    //     "d4868getDescription");
    d4868getErrMsg = (_d4868getErrMsg)GetProcAddress(hInstanceControl, "d4868getErrMsg");
    // d4868getI = (_d4868getI)GetProcAddress(hInstanceControl, "d4868getI");
    // d4868getF = (_d4868getF)GetProcAddress(hInstanceControl, "d4868getF");
    // d4868getD = (_d4868getD)GetProcAddress(hInstanceControl, "d4868getD");
    // d4868getS = (_d4868getS)GetProcAddress(hInstanceControl, "d4868getS");
    // d4868getI1D = (_d4868getI1D)GetProcAddress(hInstanceControl, "d4868getI1D");
    // d4868getF1D = (_d4868getF1D)GetProcAddress(hInstanceControl, "d4868getF1D");
    // d4868getD1D = (_d4868getD1D)GetProcAddress(hInstanceControl, "d4868getD1D");
    // d4868setI1Dentry = (_d4868setI1Dentry)GetProcAddress(hInstanceControl,
    "d4868setI1Dentry");
    // d4868setI2Dentry = (_d4868setI2Dentry)GetProcAddress(hInstanceControl,
    "d4868setI2Dentry");
    // d4868setI3Dentry = (_d4868setI3Dentry)GetProcAddress(hInstanceControl,
    "d4868setI3Dentry");
    // d4868setF1Dentry = (_d4868setF1Dentry)GetProcAddress(hInstanceControl,
    "d4868setF1Dentry");
    // d4868setF2Dentry = (_d4868setF2Dentry)GetProcAddress(hInstanceControl,
    "d4868setF2Dentry");
    // d4868setF3Dentry = (_d4868setF3Dentry)GetProcAddress(hInstanceControl,
    "d4868setF3Dentry");
    // d4868setD1Dentry = (_d4868setD1Dentry)GetProcAddress(hInstanceControl,
    "d4868setD1Dentry");
    // d4868setD2Dentry = (_d4868setD2Dentry)GetProcAddress(hInstanceControl,
    "d4868setD2Dentry");
    // d4868setD3Dentry = (_d4868setD3Dentry)GetProcAddress(hInstanceControl,
    "d4868setD3Dentry");
    // d4868setS1Dentry = (_d4868setS1Dentry)GetProcAddress(hInstanceControl,
    "d4868setS1Dentry");
    // d4868setS2Dentry = (_d4868setS2Dentry)GetProcAddress(hInstanceControl,
    "d4868setS2Dentry");
    // d4868getSeverityMax = (_d4868getSeverityMax)GetProcAddress(hInstanceControl,
    "d4868getSeverityMax");
    d4868getUnits = (_d4868getUnits)GetProcAddress(hInstanceControl, "d4868getUnits");
    d4868initProg = (_d4868initProg)GetProcAddress(hInstanceControl, "d4868initProg");
    d4868isValidParamName = (_d4868isValidParamName)GetProcAddress(hInstanceControl,
        "d4868isValidParamName");
    // d4868parseEfile = (_d4868parseEfile)GetProcAddress(hInstanceControl, "d4868parseEfile");
    // d4868parseFile = (_d4868parseFile)GetProcAddress(hInstanceControl, "d4868parseFile");
    // d4868parseString = (_d4868parseString)GetProcAddress(hInstanceControl,
    "d4868parseString");
    d4868run = (_d4868run)GetProcAddress(hInstanceControl, "d4868run");
    // d4868setDataListD = (_d4868setDataListD)GetProcAddress(hInstanceControl,
```



```

"d4868setDataListD");
// d4868setDataListF = (_d4868setDataListF)GetProcAddress(hInstanceControl,
"d4868setDataListF");
// d4868setDataListI = (_d4868setDataListI)GetProcAddress(hInstanceControl,
"d4868setDataListI");
// d4868setI = (_d4868setI)GetProcAddress(hInstanceControl, "d4868setI");
// d4868setF = (_d4868setF)GetProcAddress(hInstanceControl, "d4868setF");
// d4868setD = (_d4868setD)GetProcAddress(hInstanceControl, "d4868setD");
// d4868setS = (_d4868setS)GetProcAddress(hInstanceControl, "d4868setS");
// d4868setI1D = (_d4868setI1D)GetProcAddress(hInstanceControl, "d4868setI1D");
// d4868setF1D = (_d4868setF1D)GetProcAddress(hInstanceControl, "d4868setF1D");
// d4868setD1D = (_d4868setD1D)GetProcAddress(hInstanceControl, "d4868setD1D");
// d4868setI1Dentry = (_d4868setI1Dentry)GetProcAddress(hInstanceControl,
"d4868setI1Dentry");
// d4868setI2Dentry = (_d4868setI2Dentry)GetProcAddress(hInstanceControl,
"d4868setI2Dentry");
// d4868setI3Dentry = (_d4868setI3Dentry)GetProcAddress(hInstanceControl,
"d4868setI3Dentry");
// d4868setF1Dentry = (_d4868setF1Dentry)GetProcAddress(hInstanceControl,
"d4868setF1Dentry");
// d4868setF2Dentry = (_d4868setF2Dentry)GetProcAddress(hInstanceControl,
"d4868setF2Dentry");
// d4868setF3Dentry = (_d4868setF3Dentry)GetProcAddress(hInstanceControl,
"d4868setF3Dentry");
// d4868setD1Dentry = (_d4868setD1Dentry)GetProcAddress(hInstanceControl,
"d4868setD1Dentry");
// d4868setD2Dentry = (_d4868setD2Dentry)GetProcAddress(hInstanceControl,
"d4868setD2Dentry");
// d4868setD3Dentry = (_d4868setD3Dentry)GetProcAddress(hInstanceControl,
"d4868setD3Dentry");
// d4868setS1Dentry = (_d4868setS1Dentry)GetProcAddress(hInstanceControl,
"d4868setS1Dentry");
// d4868setS2Dentry = (_d4868setS2Dentry)GetProcAddress(hInstanceControl,
"d4868setS2Dentry");
d4868terminate = (_d4868terminate)GetProcAddress(hInstanceControl, "d4868terminate");

// Additional non-ARP 4868 functions
d4868writeStrToLog = (_d4868writeStrToLog)GetProcAddress(hInstanceControl,
"d4868writeStrToLog");
// d4868loadModel = (_d4868loadModel)GetProcAddress(hInstanceControl, "d4868loadModel");
// d4868closeModel = (_d4868closeModel)GetProcAddress(hInstanceControl, "d4868closeModel");
// d4868saveModel = (_d4868saveModel)GetProcAddress(hInstanceControl, "d4868saveModel");
// d4868showModel = (_d4868showModel)GetProcAddress(hInstanceControl, "d4868showModel");
// d4868getInputDataList = (_d4868getInputDataList)GetProcAddress(hInstanceControl,
"d4868getInputDataList");
// d4868getInputDataListSize = (_d4868getInputDataListSize)GetProcAddress(hInstanceControl,
"d4868getInputDataListSize");
// d4868programInfo = (_d4868programInfo)GetProcAddress(hInstanceControl,
"d4868programInfo");
// d4868evalFunctionI = (_d4868evalFunctionI)GetProcAddress(hInstanceControl,
"d4868evalFunctionI");
// d4868evalFunctionF = (_d4868evalFunctionF)GetProcAddress(hInstanceControl,
"d4868evalFunctionF");
// d4868evalFunctionD = (_d4868evalFunctionD)GetProcAddress(hInstanceControl,
"d4868evalFunctionD");
// d4868evalFunctionS = (_d4868evalFunctionS)GetProcAddress(hInstanceControl,
"d4868evalFunctionS");

} else {
return 0;
}
// Inint ErrorMessage
strcpy(ErrorMessage, "");

// Create a list of parameters runtime (ArrayOfStrings)
variableNumberOfElements = 6;
ArrayOfStrings = (char **)malloc(variableNumberOfElements * sizeof(char*)); // Creates 6 char
*
for(int i = 0; i < variableNumberOfElements; i++) {
ArrayOfStrings[i] = (char *)malloc((255 + 1) * sizeof(char)); // Creates 6 strings of size
255+0
}
// Fill the array to load a model with options
strcpy(ArrayofStrings[0], "-loadModel");
strcpy(ArrayofStrings[1], "TJET_API.mxl");

```




```
strcpy(ArrayofStrings[2], "-showModel");
// strcpy(ArrayofStrings[3], "true");
strcpy(ArrayofStrings[3], "false");
strcpy(ArrayofStrings[4], "-showProgress");
strcpy(ArrayofStrings[5], "true");
// strcpy(ArrayofStrings[5], "false");

// test program:
if ((*d4868initProg)(variableNumberOfElements,ArrayofStrings) != 0) {
    // There is no log file yet!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    (*d4868terminate)();
    FreeLibrary(hInstanceControl);
    exit(-1);
}

if ((*d4868activateLog)("GSPAPI.log") > 0) {
// if ((*d4868activateLog)("") > 0) {
    (*d4868getErrMsg)(&ErrorMessage[0], 256);
    (*d4868writeStrToLog)(ErrorMessage);
} else if ((*d4868activateLog)("") < 0) {
    (*d4868terminate)();
    FreeLibrary(hInstanceControl);
    exit(-1);
}

// TEST to generate message that 2nd call not allowed!
if ((*d4868initProg)(variableNumberOfElements,ArrayofStrings) != 0) {
    (*d4868getErrMsg)(&ErrorMessage[0], 256);
    (*d4868writeStrToLog)(ErrorMessage);
}

// Now free the parameter list (ArrayofStrings)
for (i = 0; i < variableNumberOfElements; i++) {
    free(ArrayofStrings[i]); // Frees all strings
}
free(ArrayofStrings); // Frees the pointer

// Create a list
PParamList = &ParamList[0];
// strcpy(DataTypeString,"DOUBLE");
strcpy(DataTypeString,"D");
DataType = DataTypeString;

// Test if d4868defineDataList fails, if so exit program
if ((*d4868defineDataList)(PParamList, 6, DataType, &ListID) < 0){
    (*d4868getErrMsg)(&ErrorMessage[0], 256);
    (*d4868writeStrToLog)(ErrorMessage);
    (*d4868terminate)();
    FreeLibrary(hInstanceControl);
    exit(-1);
}

strcpy(DataTypeString,"F");
DataType = DataTypeString;
(*d4868defineDataList)(&ParamList1[0], 6, DataType, &ListID1);

strcpy(DataTypeString,"I");
DataType = DataTypeString;
(*d4868defineDataList)(&ParamList2[0], 6, DataType, &ListID2);

strcpy(UnitString,"");
for (i = 0; i < 6; i++) {
    (*d4868getUnits)(ParamList[i],&UnitString[0],20);
    // printf("%s %s\t",ParamList[i],UnitString);
    // (*d4868getUnits)(ParamList1[i],&UnitString[0],20);
    // printf("%s %s\t",ParamList1[i],UnitString);
    // (*d4868getUnits)(ParamList2[i],&UnitString[0],20);
    // printf("%s %s\n",ParamList2[i],UnitString);
    (*d4868isValidParamName)(ParamList[i],&IsValid);
    if (IsValid > 0) {
```




```

        printf("%s %s\n",ParamList[i],UnitString);
    } else {
        printf("'%' does not exist in model!\n",ParamList[i]);
    }
}

printf("Running Model!\n");
// Run model
(*d4868run)();

for (i = 0; i < 6; i++) {
    (*d4868getUnits)(ParamList[i],&UnitString[0],20);
    printf("%s %s\n",ParamList[i],UnitString);
}

variableNumberOfElements = 6;
DArrayData = (double *)malloc(variableNumberOfElements * sizeof(double)); // Creates 6 doubles
FArrayData = (float *)malloc(variableNumberOfElements * sizeof(float)); // Creates 6 floats
IArrayData = (int *)malloc(variableNumberOfElements * sizeof(int)); // Creates 6
integers

for (i = 0; i < variableNumberOfElements; i++) {
    DArrayData[i]=-999.999999;
    FArrayData[i]=-999.999;
    IArrayData[i]=-999;
}
printf("\n");

(*d4868getDataListD) (ListID,6,&DArrayData[0]);
for (i = 0; i < 6; i++) {
    printf("%g\t",DArrayData[i]);
}
printf("\n");

(*d4868getDataListF) (ListID1,6,&FArrayData[0]);
for (i = 0; i < 6; i++) {
    printf("%f\t",FArrayData[i]);
}
printf("\n");

(*d4868getDataListI) (ListID2,6,&IArrayData[0]);
for (i = 0; i < 6; i++) {
    printf("%d\t",IArrayData[i]);
}
printf("\n");

// Free memory
free(DArrayData);
free(FArrayData);
free(IArrayData);

// Close the log file, if not done explicitly, then it will be always closed
// by "d4868terminate"
(*d4868closeLog)();

// Terminate engine program
(*d4868terminate)();

// Frees the loaded dynamic-link library (DLL) module and, if necessary,
// decrements its reference count. When the reference count reaches zero,
// the module is unloaded from the address space of the calling process and
// the handle is no longer valid.
FreeLibrary(hInstanceControl);

system("pause");
return 0;
}

```

Application screen shot

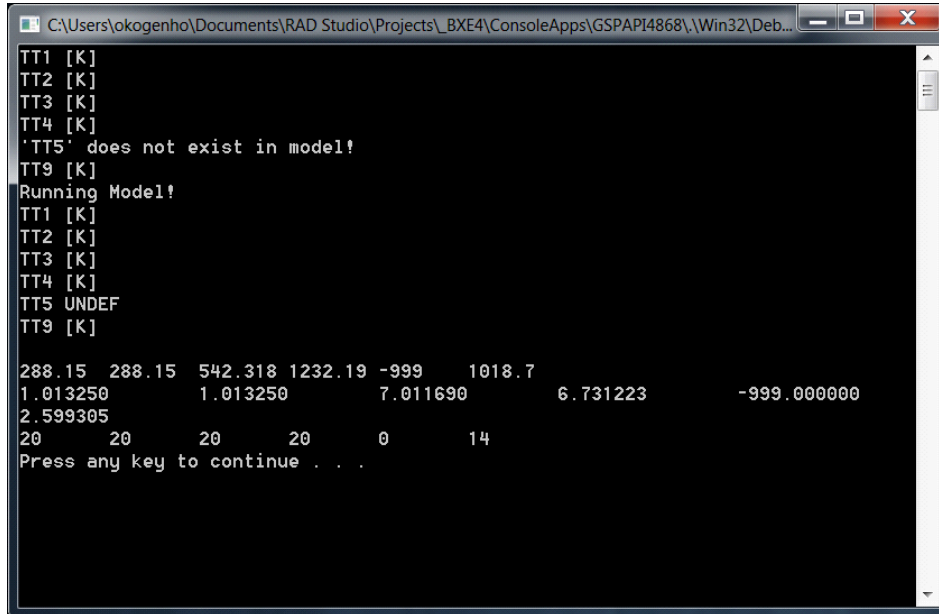


Figure 8 – Console application screenshot at finish of C-code application run

5.3.1.2 Console application in Delphi

Application source code

```

program GSPAPI4868_console;

{$APPTYPE CONSOLE}
{$DEFINE CDECL}

{$R *.res}

uses
  System.SysUtils,
  System.AnsiStrings,
  Winapi.Windows,
  Vcl.Dialogs;

type
  THandle = Integer;

// -----
// Function type definition
// -----
// 1 x4868activateLog: Activates API Function Logging
Td4868activateLog = function (const filename: PAnsiChar): Integer; cdecl;
// 2 x4868closeLog: Closes API Function Logging
Td4868closeLog = function (): Integer; cdecl;
// 3 x4868defineDataList: Define a List of Parameters to Set or Get Later
Td4868defineDataList = function (const paramList: PAnsiChar; size: Integer; const
dataType: PAnsiChar; listID: PInteger): Integer; cdecl;
// 4 x4868getArraySize[123]D: Get Size of Dimensions in Array Parameters
Td4868getArraySize1D = function (const paramName: PAnsiChar; numX: PInteger): Integer;
cdecl;
Td4868getArraySize2D = function (const paramName: PAnsiChar; numX: PInteger; numY:
PInteger): Integer; cdecl;
Td4868getArraySize3D = function (const paramName: PAnsiChar; numX: PInteger; numY:
PInteger; numZ: PInteger): Integer; cdecl;
// 5 x4868getDataList[IFD]: Get an Array of Values Associated with a List of Parameters
Td4868getDataListD = function (listID: Integer; size: Integer; arrayValues: PInteger):
Integer; cdecl;
Td4868getDataListF = function (listID: Integer; size: Integer; arrayValues: PSingle):
Integer; cdecl;
Td4868getDataListI = function (listID: Integer; size: Integer; arrayValues: PDouble):
Integer; cdecl;

```



```

// 6 x4868getDataType: Get the Data Type Attribute for the Named Parameter
Td4868getDataType = function (const paramName: PAnsiChar; dataType: PAnsiChar; buffSize
: Integer): Integer; cdecl;
// 7 x4868getDescription: Get the Description Attribute for the Named Parameter
Td4868getDescription = function (const paramName: PAnsiChar; descr: PAnsiChar; buffSize:
Integer): Integer; cdecl;
// 8 x4868getErrMsg: Get Function Error Description String
Td4868getErrMsg = function (errorMessage: PAnsiChar; buffSize: Integer): Integer;
cdecl;
// 9 x4868get[IFDS]: Get a Scalar Parameter Value
Td4868getI = function (const paramName: PAnsiChar; value: PInteger): Integer;
cdecl;
Td4868getF = function (const paramName: PAnsiChar; value: PSingle): Integer;
cdecl;
Td4868getD = function (const paramName: PAnsiChar; value: PDouble): Integer;
cdecl;
Td4868gets = function (const paramName: PAnsiChar; value: PAnsiChar; buffSize:
Integer): Integer; cdecl;
// 10 x4868get[IFD]ID: GetID Array Parameter Values
Td4868getIID = function (const paramName: PAnsiChar; arrayValues: PInteger; numGet
: Integer): Integer; cdecl;
Td4868getFID = function (const paramName: PAnsiChar; arrayValues: PSingle; numGet:
Integer): Integer; cdecl;
Td4868getDID = function (const paramName: PAnsiChar; arrayValues: PDouble; numGet:
Integer): Integer; cdecl;
// 11 x4868get[IFDS][123]Dentry: Get Array Parameter Entry Value
Td4868getIIDentry = function (const paramName: PAnsiChar; index1: Integer; value:
PInteger): Integer; cdecl;
Td4868getI2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; value: PInteger): Integer; cdecl;
Td4868getI3Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; index3: Integer; value: PInteger): Integer; cdecl;
Td4868getF1Dentry = function (const paramName: PAnsiChar; index1: Integer; value:
PSingle): Integer; cdecl;
Td4868getF2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; value: PSingle): Integer; cdecl;
Td4868getF3Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; index3: Integer; value: PSingle): Integer; cdecl;
Td4868getD1Dentry = function (const paramName: PAnsiChar; index1: Integer; value:
PDouble): Integer; cdecl;
Td4868getD2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; value: PDouble): Integer; cdecl;
Td4868getD3Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; index3: Integer; value: PDouble): Integer; cdecl;
Td4868gets1Dentry = function (const paramName: PAnsiChar; index1: Integer; value:
PAnsiChar; buffSize: Integer): Integer; cdecl;
Td4868gets2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; value: PAnsiChar; buffSize: Integer): Integer; cdecl;
// 12 x4868getSeverityMax: Get the Max Severity from Engine Program
Td4868getSeverityMax = function (severityMax: PInteger): Integer; cdecl;
// 13 x4868getUnits: Get the Units Attribute for the Named Parameter
Td4868getUnits = function (const paramName: PAnsiChar; units: PAnsiChar; buffSize:
Integer): Integer; cdecl;
// 14 x4868initProg: Initialize the Engine Program
Td4868initProg = function (numArgs: Integer; args: PPAnsiChar): Integer; cdecl;
// 15 x4868isValidParamName: Confirms If Parameter is Present in Engine Program
Td4868isValidParamName = function (const paramName: PAnsiChar; valid: PInteger): Integer;
cdecl;
// 16 x4868parseEfile: Parse an Encrypted File
Td4868parseEfile = function (const fileName: PAnsiChar): Integer; cdecl;
// 17 x4868parseFile: Parse a Plain Text File
Td4868parseFile = function (const fileName: PAnsiChar): Integer; cdecl;
// 18 x4868parseString: Parse a String
Td4868parseString = function (const str: PAnsiChar): Integer; cdecl;
// 19 x4868run: Execute the Engine Program
Td4868run = function (): Integer; cdecl;
// 20 x4868setDataList[IFD]: Set a List of Engine Program Parameters Values
Td4868setDataListD = function (listID: Integer; size: Integer; arrayValues: PDouble):
Integer; cdecl;
Td4868setDataListF = function (listID: Integer; size: Integer; arrayValues: PSingle):
Integer; cdecl;
Td4868setDataListI = function (listID: Integer; size: Integer; arrayValues: PInteger):
Integer; cdecl;
// 21 x4868set[IFDS]: Set a Scalar Parameter Value
Td4868setI = function (const paramName: PAnsiChar; value: Integer): Integer;
cdecl;
Td4868setF = function (const paramName: PAnsiChar; value: Single): Integer;
cdecl;
Td4868setD = function (const paramName: PAnsiChar; value: Double): Integer;
cdecl;
Td4868sets = function (const paramName: PAnsiChar; const value: PAnsiChar):
Integer; cdecl;
// 22 x4868set[IFD]ID: Set ID Array Parameter Values
Td4868setIID = function (const paramName: PAnsiChar; arrayValues: PInteger; numSet
: Integer): Integer; cdecl;
Td4868setFID = function (const paramName: PAnsiChar; arrayValues: PSingle; numSet:
Integer): Integer; cdecl;

```



```

Td4868setD1D      = function (const paramName: PAnsiChar; arrayValues: PDouble; numSet:
Integer): Integer; cdecl;
// 23 x4868set[IFDS][123]Dentry: Set an Array Parameter Entry Value
Td4868setI1Dentry = function (const paramName: PAnsiChar; index1: Integer; value:
Integer): Integer; cdecl;
Td4868setI2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; value: Integer): Integer; cdecl;
Td4868setI3Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; index3: Integer; value: Integer): Integer; cdecl;
Td4868setF1Dentry = function (const paramName: PAnsiChar; index1: Integer; value:
Single): Integer; cdecl;
Td4868setF2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; value: Single): Integer; cdecl;
Td4868setF3Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; index3: Integer; value: Single): Integer; cdecl;
Td4868setD1Dentry = function (const paramName: PAnsiChar; index1: Integer; value:
Double): Integer; cdecl;
Td4868setD2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; value: Double): Integer; cdecl;
Td4868setD3Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; index3: Integer; value: Double): Integer; cdecl;
Td4868sets1Dentry = function (const paramName: PAnsiChar; index1: Integer; const value
: PAnsiChar): Integer; cdecl;
Td4868sets2Dentry = function (const paramName: PAnsiChar; index1: Integer; index2:
Integer; const value: PAnsiChar): Integer; cdecl;
// 24 x4868terminate: Instructs Engine Program to Terminate
Td4868terminate   = function (): Integer; cdecl;

// Additional non-ARP 4868 functions
// a.1 x4868writeStrToLog: Write String to Log File
Td4868writeStrToLog = function (const LogString: PAnsiChar): Integer; cdecl;
// a.2 x4868loadModel: Instructs Engine Program to Load a Model
Td4868loadModel     = function (numArgs: Integer; args: PPAnsiChar): Integer; cdecl;
// a.3 x4868closeModel: Instructs Engine Program to Close Model
Td4868closeModel    = function (numArgs: Integer; args: PPAnsiChar): Integer; cdecl;
// a.4 x4868saveModel: Instructs Engine Program to Save Model
Td4868saveModel     = function (numArgs: Integer; args: PPAnsiChar): Integer; cdecl;
// a.5 x4868showModel: Instructs Engine Program to Show Model
Td4868showModel     = function (numArgs: Integer; args: PPAnsiChar): Integer; cdecl;
// a.6 x4868getInputDataList: Get a List of Input Parameters to Set Later
Td4868getInputDataList = function (paramList: PPAnsiChar; const size: Integer; dataType:
PAnsiChar; listID: PInteger): Integer; cdecl;
// a.7 x4868getInputDataListSize: Get List Size of Input Parameters List
Td4868getInputDataListSize = function (listSize, maxSizeString: PInteger): Integer; cdecl;
// a.8 x4868programInfo: Get Program Information
Td4868programInfo      = function (progName, progVersion : PAnsiChar): Integer; cdecl;
// a.9 x4868evalFunction[IFDS]: Get a Scalar Return Value From a Function Call
Td4868evalFunctionI    = function (const paramName: PAnsiChar; value: PInteger): Integer;
cdecl;
Td4868evalFunctionF    = function (const paramName: PAnsiChar; value: PSingle): Integer;
cdecl;
Td4868evalFunctionD    = function (const paramName: PAnsiChar; value: PDouble): Integer;
cdecl;
Td4868evalFunctions    = function (const paramName: PAnsiChar; value: PAnsiChar; buffSize:
Integer): Integer; cdecl;
// -----

var
dllHandle      : THandle;

// -----
// Function variable declaration
// -----
// 1 x4868activateLog: Activates API Function Logging
d4868activateLog : Td4868activateLog;
// 2 x4868closeLog: Closes API Function Logging
d4868closeLog    : Td4868closeLog;
// 3 x4868defineDataList: Define a List of Parameters to Set or Get Later
d4868defineDataList : Td4868defineDataList;
// 4 x4868getArraySize[123]D: Get Size of Dimensions in Array Parameters
d4868getArraySize1D : Td4868getArraySize1D;
d4868getArraySize2D : Td4868getArraySize2D;
d4868getArraySize3D : Td4868getArraySize3D;
// 5 x4868getDataList[IFD]: Get an Array of Values Associated with a List of Parameters
d4868getDataListD  : Td4868getDataListD;
d4868getDataListF  : Td4868getDataListF;
d4868getDataListI  : Td4868getDataListI;
// 6 x4868getDataType: Get the Data Type Attribute for the Named Parameter
d4868getDataType   : Td4868getDataType;
// 7 x4868getDescription: Get the Description Attribute for the Named Parameter
d4868getDescription : Td4868getDescription;
// 8 x4868getErrMsg: Get Function Error Description String
d4868getErrMsg      : Td4868getErrMsg;
// 9 x4868get[IFDS]: Get a Scalar Parameter Value
d4868getI           : Td4868getI;

```



```

d4868getF : Td4868getF;
d4868getD : Td4868getD;
d4868getS : Td4868getS;
// 10 x4868get[IFD]ID: GetID Array Parameter Values
d4868getI1D : Td4868getI1D;
d4868getF1D : Td4868getF1D;
d4868getD1D : Td4868getD1D;
// 11 x4868get[IFDS][123]Dentry: Get Array Parameter Entry Value
d4868getI1Dentry : Td4868getI1Dentry;
d4868getI2Dentry : Td4868getI2Dentry;
d4868getI3Dentry : Td4868getI3Dentry;
d4868getF1Dentry : Td4868getF1Dentry;
d4868getF2Dentry : Td4868getF2Dentry;
d4868getF3Dentry : Td4868getF3Dentry;
d4868getD1Dentry : Td4868getD1Dentry;
d4868getD2Dentry : Td4868getD2Dentry;
d4868getD3Dentry : Td4868getD3Dentry;
d4868getS1Dentry : Td4868getS1Dentry;
d4868getS2Dentry : Td4868getS2Dentry;
// 12 x4868getSeverityMax: Get the Max Severity from Engine Program
d4868getSeverityMax : Td4868getSeverityMax;
// 13 x4868getUnits: Get the Units Attribute for the Named Parameter
d4868getUnits : Td4868getUnits;
// 14 x4868initProg: Initialize the Engine Program
d4868initProg : Td4868initProg;
// 15 x4868isValidParamName: Confirms If Parameter is Present in Engine Program
d4868isValidParamName : Td4868isValidParamName;
// 16 x4868parseEfile: Parse an Encrypted File
d4868parseEfile : Td4868parseEfile;
// 17 x4868parseFile: Parse a Plain Text File
d4868parseFile : Td4868parseFile;
// 18 x4868parseString: Parse a String
d4868parseString : Td4868parseString;
// 19 x4868run Execute the Engine Program
d4868run : Td4868run;
// 20 x4868setDataList[IFD]: Set a List of Engine Program Parameters Values
d4868setDataListD : Td4868setDataListD;
d4868setDataListF : Td4868setDataListF;
d4868setDataListI : Td4868setDataListI;
// 21 x4868set[IFDS]: Set a Scalar Parameter Value
d4868setI : Td4868setI;
d4868setF : Td4868setF;
d4868setD : Td4868setD;
d4868sets : Td4868sets;
// 22 x4868set[IFD]ID: Set ID Array Parameter Values
d4868setI1D : Td4868setI1D;
d4868setF1D : Td4868setF1D;
d4868setD1D : Td4868setD1D;
// 23 x4868set[IFDS][123]Dentry: Set an Array Parameter Entry Value
d4868setI1Dentry : Td4868setI1Dentry;
d4868setI2Dentry : Td4868setI2Dentry;
d4868setI3Dentry : Td4868setI3Dentry;
d4868setF1Dentry : Td4868setF1Dentry;
d4868setF2Dentry : Td4868setF2Dentry;
d4868setF3Dentry : Td4868setF3Dentry;
d4868setD1Dentry : Td4868setD1Dentry;
d4868setD2Dentry : Td4868setD2Dentry;
d4868setD3Dentry : Td4868setD3Dentry;
d4868sets1Dentry : Td4868sets1Dentry;
d4868sets2Dentry : Td4868sets2Dentry;
// 24 x4868terminate: Instructs Engine Program to Terminate
d4868terminate : Td4868terminate;

// Additional non-ARP 4868 functions
// a.1 x4868writeToLog: Write String to Log File
d4868writeStrToLog : Td4868writeStrToLog;
// a.2 x4868loadModel: Instructs Engine Program to Load a Model
d4868loadModel : Td4868loadModel;
// a.3 x4868closeModel: Instructs Engine Program to Close Model
d4868closeModel : Td4868closeModel;
// a.4 x4868saveModel: Instructs Engine Program to Save Model
d4868saveModel : Td4868saveModel;
// a.5 x4868showModel: Instructs Engine Program to Show Model
d4868showModel : Td4868showModel;
// a.6 x4868getInputDataList: Get a List of Input Parameters to Set Later
d4868getInputDataList : Td4868getInputDataList;
// a.7 x4868getInputDataListSize: Get List Size of Input Parameters List
d4868getInputDataListSize : Td4868getInputDataListSize;
// a.8 x4868programInfo: Get Program Information
d4868programInfo : Td4868programInfo;
// a.9 x4868evalFunction[IFDS]: Get a Scalar Return Value From a Function Call
d4868evalFunctionI : Td4868evalFunctionI;
d4868evalFunctionF : Td4868evalFunctionF;
d4868evalFunctionD : Td4868evalFunctionD;
d4868evalFunctions : Td4868evalFunctions;
// -----

```



```
PArgumentsList,
PParamList,
PInputList          : PPAnsiChar;
PUnitString         : PAnsiChar;
DataType, UnitString,
DataTypeString,
aTmpString,
aLogMessage,
GSPName, GSPVersion : AnsiString;
ArgumentList,
ParamList,
InputList,
UnitStringList      : array of AnsiString;
i, ListID, isValid,
inputListID, intResult,
maxSize,
maxStringLength     : Integer;
DArrayData          : array of Double;

function Pause(): Integer;
begin
    writeLn ('Press ENTER key to continue...');
    readLn;
    Result := 1;
end;

{$POINTERMATH ON}
begin
    dllHandle := 0;
    try
        // Load the GSP DLL library
        try
            // Load the dll
            dllHandle := LoadLibrary('GSP.dll') ;
            if dllHandle <> 0 then
                begin
                    // -----
                    // Function hook to DLL function address
                    // -----
                    // 1 x4868activateLog: Activates API Function Logging
                    @d4868activateLog := GetProcAddress(dllHandle, 'd4868activateLog');
                    // 2 x4868closeLog: Closes API Function Logging
                    @d4868closeLog := GetProcAddress(dllHandle, 'd4868closeLog');
                    // 3 x4868defineDataList: Define a List of Parameters to Set or Get Later
                    @d4868defineDataList := GetProcAddress(dllHandle, 'd4868defineDataList');
                    // 4 x4868getArraySize[123]D: Get Size of Dimensions in Array Parameters
                    @d4868getArraySize1D := GetProcAddress(dllHandle, 'd4868getArraySize1D');
                    @d4868getArraySize2D := GetProcAddress(dllHandle, 'd4868getArraySize2D');
                    @d4868getArraySize3D := GetProcAddress(dllHandle, 'd4868getArraySize3D');
                    // 5 x4868getDataList[IFD]: Get an Array of Values Associated with a List of Parameters
                    @d4868getDataListD := GetProcAddress(dllHandle, 'd4868getDataListD');
                    @d4868getDataListF := GetProcAddress(dllHandle, 'd4868getDataListF');
                    @d4868getDataListI := GetProcAddress(dllHandle, 'd4868getDataListI');
                    // 6 x4868getDataType: Get the Data Type Attribute for the Named Parameter
                    @d4868getDataType := GetProcAddress(dllHandle, 'd4868getDataType');
                    // 7 x4868getDescription: Get the Description Attribute for the Named Parameter
                    @d4868getDescription := GetProcAddress(dllHandle, 'd4868getDescription');
                    // 8 x4868getErrMsg: Get Function Error Description String
                    @d4868getErrMsg := GetProcAddress(dllHandle, 'd4868getErrMsg');
                    // 9 x4868get[IFDS]: Get a Scalar Parameter Value
                    @d4868getI := GetProcAddress(dllHandle, 'd4868getI');
                    @d4868getF := GetProcAddress(dllHandle, 'd4868getF');
                    @d4868getD := GetProcAddress(dllHandle, 'd4868getD');
                    @d4868gets := GetProcAddress(dllHandle, 'd4868gets');
                    // 10 x4868get[IFD]1D: Get1D Array Parameter Values
                    @d4868getI1D := GetProcAddress(dllHandle, 'd4868getI1D');
                    @d4868getF1D := GetProcAddress(dllHandle, 'd4868getF1D');
                    @d4868getD1D := GetProcAddress(dllHandle, 'd4868getD1D');
                    // 11 x4868get[IFDS][123]Dentry: Get Array Parameter Entry Value
                    @d4868setI1Dentry := GetProcAddress(dllHandle, 'd4868setI1Dentry');
                    @d4868setI2Dentry := GetProcAddress(dllHandle, 'd4868setI2Dentry');
                    @d4868setI3Dentry := GetProcAddress(dllHandle, 'd4868setI3Dentry');
                    @d4868setF1Dentry := GetProcAddress(dllHandle, 'd4868setF1Dentry');
                    @d4868setF2Dentry := GetProcAddress(dllHandle, 'd4868setF2Dentry');
                    @d4868setF3Dentry := GetProcAddress(dllHandle, 'd4868setF3Dentry');
                    @d4868setD1Dentry := GetProcAddress(dllHandle, 'd4868setD1Dentry');
                    @d4868setD2Dentry := GetProcAddress(dllHandle, 'd4868setD2Dentry');
                    @d4868setD3Dentry := GetProcAddress(dllHandle, 'd4868setD3Dentry');
                    @d4868setS1Dentry := GetProcAddress(dllHandle, 'd4868setS1Dentry');
                    @d4868setS2Dentry := GetProcAddress(dllHandle, 'd4868setS2Dentry');
                    // 12 x4868getSeverityMax: Get the Max Severity from Engine Program
                    @d4868getSeverityMax := GetProcAddress(dllHandle, 'd4868getSeverityMax');
                    // 13 x4868getUnits: Get the Units Attribute for the Named Parameter
                    @d4868getUnits := GetProcAddress(dllHandle, 'd4868getUnits');
```




```
// 14 x4868initProg: Initialize the Engine Program
@d4868initProg := GetProcAddress(dllHandle, 'd4868initProg');
// 15 x4868isValidParamName: Confirms If Parameter is Present in Engine Program
@d4868isValidParamName := GetProcAddress(dllHandle, 'd4868isValidParamName');
// 16 x4868parseEfile: Parse an Encrypted File
@d4868parseEfile := GetProcAddress(dllHandle, 'd4868parseEfile');
// 17 x4868parseFile: Parse a Plain Text File
@d4868parseFile := GetProcAddress(dllHandle, 'd4868parseFile');
// 18 x4868parseString: Parse a String
@d4868parseString := GetProcAddress(dllHandle, 'd4868parseString');
// 19 x4868run Execute the Engine Program
@d4868run := GetProcAddress(dllHandle, 'd4868run');
// 20 x4868setDataList[IFD]: Set a List of Engine Program Parameters Values
@d4868setDataListD := GetProcAddress(dllHandle, 'd4868setDataListD');
@d4868setDataListF := GetProcAddress(dllHandle, 'd4868setDataListF');
@d4868setDataListI := GetProcAddress(dllHandle, 'd4868setDataListI');
// 21 x4868set[IFDS]: Set a Scalar Parameter Value
@d4868setI := GetProcAddress(dllHandle, 'd4868setI');
@d4868setF := GetProcAddress(dllHandle, 'd4868setF');
@d4868setD := GetProcAddress(dllHandle, 'd4868setD');
@d4868setS := GetProcAddress(dllHandle, 'd4868setS');
// 22 x4868set[IFD]ID: Set ID Array Parameter Values
@d4868setIID := GetProcAddress(dllHandle, 'd4868setIID');
@d4868setFID := GetProcAddress(dllHandle, 'd4868setFID');
@d4868setDID := GetProcAddress(dllHandle, 'd4868setDID');
// 23 x4868set[IFDS][123]Dentry: Set an Array Parameter Entry Value
@d4868setIIDentry := GetProcAddress(dllHandle, 'd4868setIIDentry');
@d4868setI2Dentry := GetProcAddress(dllHandle, 'd4868setI2Dentry');
@d4868setI3Dentry := GetProcAddress(dllHandle, 'd4868setI3Dentry');
@d4868setF1Dentry := GetProcAddress(dllHandle, 'd4868setF1Dentry');
@d4868setF2Dentry := GetProcAddress(dllHandle, 'd4868setF2Dentry');
@d4868setF3Dentry := GetProcAddress(dllHandle, 'd4868setF3Dentry');
@d4868setD1Dentry := GetProcAddress(dllHandle, 'd4868setD1Dentry');
@d4868setD2Dentry := GetProcAddress(dllHandle, 'd4868setD2Dentry');
@d4868setD3Dentry := GetProcAddress(dllHandle, 'd4868setD3Dentry');
@d4868setS1Dentry := GetProcAddress(dllHandle, 'd4868setS1Dentry');
@d4868setS2Dentry := GetProcAddress(dllHandle, 'd4868setS2Dentry');
// 24 x4868terminate: Instructs Engine Program to Terminate
@d4868terminate := GetProcAddress(dllHandle, 'd4868terminate');

// Additional non-ARP 4868 functions
// a.1 x4868writeToLog: Write String to Log File
@d4868writeStrToLog := GetProcAddress(dllHandle, 'd4868writeStrToLog');
// a.2 x4868loadModel: Instructs Engine Program to Load a Model
@d4868loadModel := GetProcAddress(dllHandle, 'd4868loadModel');
// a.3 x4868closeModel: Instructs Engine Program to Close Model
@d4868closeModel := GetProcAddress(dllHandle, 'd4868closeModel');
// a.4 x4868saveModel: Instructs Engine Program to Save Model
@d4868saveModel := GetProcAddress(dllHandle, 'd4868saveModel');
// a.5 x4868showModel: Instructs Engine Program to Show Model
@d4868showModel := GetProcAddress(dllHandle, 'd4868showModel');
// a.6 x4868getInputDataList: Get a List of Input Parameters to Set Later
@d4868getInputDataList := GetProcAddress(dllHandle, 'd4868getInputDataList');
// a.7 x4868getInputDataListSize: Get List Size of Input Parameters List
@d4868getInputDataListSize := GetProcAddress(dllHandle, 'd4868getInputDataListSize');
// a.8 x4868programInfo: Get Program Information
@d4868programInfo := GetProcAddress(dllHandle, 'd4868programInfo');
// a.9 x4868evalFunction[IFDS]: Get a Scalar Return Value From a Function Call
@d4868evalFunctionI := GetProcAddress(dllHandle, 'd4868evalFunctionI');
@d4868evalFunctionF := GetProcAddress(dllHandle, 'd4868evalFunctionF');
@d4868evalFunctionD := GetProcAddress(dllHandle, 'd4868evalFunctionD');
@d4868evalFunctionsS := GetProcAddress(dllHandle, 'd4868evalFunctionsS');
// -----

// -----
// Console program
// -----

// initProg
SetLength(ArgumentList,8); // will create an array of 4 AnsiStrings
ArgumentList[0] := '-loadModel';
ArgumentList[1] := 'TJET_API.mx1';
ArgumentList[2] := '-showmodel';
// ArgumentList[3] := 'true';
ArgumentList[3] := 'false';
// ArgumentList[4] := '-showProgress';
ArgumentList[4] := 'false';
// ArgumentList[5] := 'true';
ArgumentList[5] := 'false';
ArgumentList[6] := '-createLog';
ArgumentList[7] := 'GSPAPI_DXE5';
PArgumentsList := @ArgumentList[0];

d4868initProg(High(ArgumentList)+1, PArgumentsList);
```



```
(* Logfile already loaded in d4868initProg, so not necessary!
// activateLog
d4868activateLog('GSPAPI_DXE5');
// OR
//d4868activateLog('');
*)

// Get program info!
SetLength(GSPName, 35);
SetLength(GSPversion, 25);
d4868programInfo(PAnsiChar(GSPName), PAnsiChar(GSPversion));
aLogMessage := Format('%s : %s', [PAnsiChar(GSPName), PAnsiChar(GSPversion)]);
writeln(PAnsiChar(aLogMessage));
d4868writeStrToLog(PAnsiChar(aLogMessage));

(* Model already loaded in d4868initProg, so not necessary!
// loadModel
SetLength(ArgumentList, 6); // will create an array of 3 AnsiStrings
ArgumentList[0] := '-fileName';
ArgumentList[1] := 'TJET_API.mxl';
ArgumentList[2] := '-showModel';
ArgumentList[3] := 'false';
//ArgumentList[3] := 'true';
ArgumentList[4] := '-showProgress';
ArgumentList[5] := 'true';
//ArgumentList[5] := 'false';
PArgumentsList := @ArgumentList[0];
d4868loadModel(6, PArgumentsList);
*)

// Parse function string I "GSPeditComponent"
// Execute a published method of the model
SetLength(aLogMessage, 1024);
aLogMessage := 'ExecFunction GSPeditComponent Compressor';
d4868evalFunctionI(PAnsiChar(aLogMessage), @intResult);
case intResult of
0:begin
aLogMessage := Format('d4868evalFunctionI(%s) = 'FALSE'', [aLogMessage]);
end;
1:begin
aLogMessage := Format('d4868evalFunctionI(%s) = 'TRUE'', [aLogMessage]);
end;
else
aLogMessage := Format('d4868evalFunctionI(%s) = 'UNDEFINED'', [aLogMessage]);
end;
writeln(aLogMessage);

// Parse function string II "GSPGetNrOfComponents"
SetLength(aLogMessage, 1024);
// Execute a published method of the model
aLogMessage := 'ExecFunction GSPGetNrOfComponents';
d4868evalFunctionI(PAnsiChar(aLogMessage), @intResult);
if intResult > 0 then
aLogMessage := Format('d4868evalFunctionI(%s) = '%d'', [aLogMessage, intResult])
else
aLogMessage := Format('d4868evalFunctionI(%s) = 'UNDEFINED'', [aLogMessage]);
writeln(aLogMessage);

// Parse function string III "GSPGetComponentNames"
SetLength(aLogMessage, 1024);
// Execute a published method of the model
aLogMessage := 'ExecFunction GSPGetComponentNames';
SetLength(aTmpString, 1024);
d4868evalFunctionS(PAnsiChar(aLogMessage), PAnsiChar(aTmpString), 1024);
if aTmpString <> '' then
aLogMessage := Format('d4868evalFunctionI(%s) = '%s'', [aLogMessage, Trim(
StringReplace(aTmpString, #13#10, ' ', [rfReplaceAll])]);
writeln(aLogMessage);

// defineDataList
SetLength(ParamList, 6); // will create an array of 6 AnsiStrings
SetLength(DArrayData, 6);
SetLength(UnitStringList, 6);
for i := 0 to length(ParamList)-1 do
begin
ParamList[i] := 'T'+IntToStr(i+1);
DArrayData[i] := 0.0;
SetLength(UnitStringList[i], 20);
end;
ParamList[5] := 'T9';
PParamList := @ParamList[0];
DataType := 'D';
d4868defineDataList(PParamList, 6, PAnsiChar(DataType), @ListID);
```




```

// TEST
aLogMessage := Format('Defined a data set; ID = %d',[listID]);
writeln(aLogMessage);

// Input parameters?
d4868getInputDataListSize(@maxSize, @maxStringLength);
aLogMessage := Format('Parameters in input data set: %d (max length: %d)',[maxSize,
maxStringLength]);
writeln(aLogMessage);

// Reserve memory
SetLength(InputList,maxSize); // will create an array of 'maxSize' AnsiStrings
// Now set the length of each element!
for i := 0 to maxSize-1 do
begin
SetLength(InputList[i],maxStringLength);
//InputList[i] := ''; // NO!!!!!!
end;

PInputList := @InputList[0];
d4868getInputDataList(PInputList, maxSize, PAnsiChar(DataType), @inputListID);
aLogMessage := Format('Nr. of input parameters: %d; listID : %d',[maxSize,inputListID
]);
writeln(aLogMessage);
for i := 0 to maxSize-1 do
begin
// Print data values
//d4868getUnits(PParamList[i],PUnitString,20);
aLogMessage := Format('%d: %s',[i, PAnsiChar(InputList[i])]); // Essential to use
PAnsiChar! can be junk chars left in other chars of string[20]
//aLogMessage := Format('%d: %s',[i, InputList[i]]); // NO, do NOT do this!!!
writeln(aLogMessage);
end;

// getUnits
SetLength(UnitString,20);
SetLength(DataTypeString,20);
PUnitString := PAnsiChar(UnitString);
for i := 0 to 6-1 do
begin
//d4868getUnits(PParamList[i],@UnitString,20); // Gives -> Runtime error 216!!! DON'T
DO THIS!!!
d4868getUnits(PParamList[i],PAnsiChar(UnitString),20); // Correct!!!
System.AnsiStrings.StrCopy(PAnsiChar(UnitStringList[i]), PAnsiChar(UnitString));
if d4868getDataType(PParamList[i],PAnsiChar(DataTypeString),20) > 0 then
begin
// report!
SetLength(aLogMessage,1024);
d4868getErrMsg(PAnsiChar(aLogMessage), 1024);
d4868writeStrToLog(PAnsiChar(aLogMessage));
end;
d4868isValidParamName(PParamList[i],@IsValid);
if (IsValid > 0) then
//aLogMessage := Format('%s %s',[PParamList[i],PAnsiChar(UnitString)])
aLogMessage := Format('%s %s %s',[PParamList[i],PAnsiChar(UnitStringList[i]),
PAnsiChar(DataTypeString)])
else
aLogMessage := Format('%s does not exist in model!',[PParamList[i]]);
writeln(aLogMessage);
end;

// run
d4868run();

// getDataListD
d4868getDataListD(ListID,6,@DArrayData[0]);
for i := 0 to 6-1 do
begin
// Print data values
//d4868getUnits(PParamList[i],PUnitString,20);
aLogMessage := Format('%s %f %s',[PParamList[i],DArrayData[i],UnitStringList[i]]);
writeln(aLogMessage);
end;

// closeModel, not required as d4868terminate also closes the model
d4868closeModel(0, nil);

// closeLog
d4868closeLog();

```



```
// terminate
d4868terminate();

end;
finally
// Always finish with closing the library to prevent open handles/memory leaks
if dllHandle <> 0 then FreeLibrary(dllHandle);
end;

except
on E: Exception do
writeln(E.ClassName, ': ', E.Message);
end;
Pause();
end.
```

Application screen shot

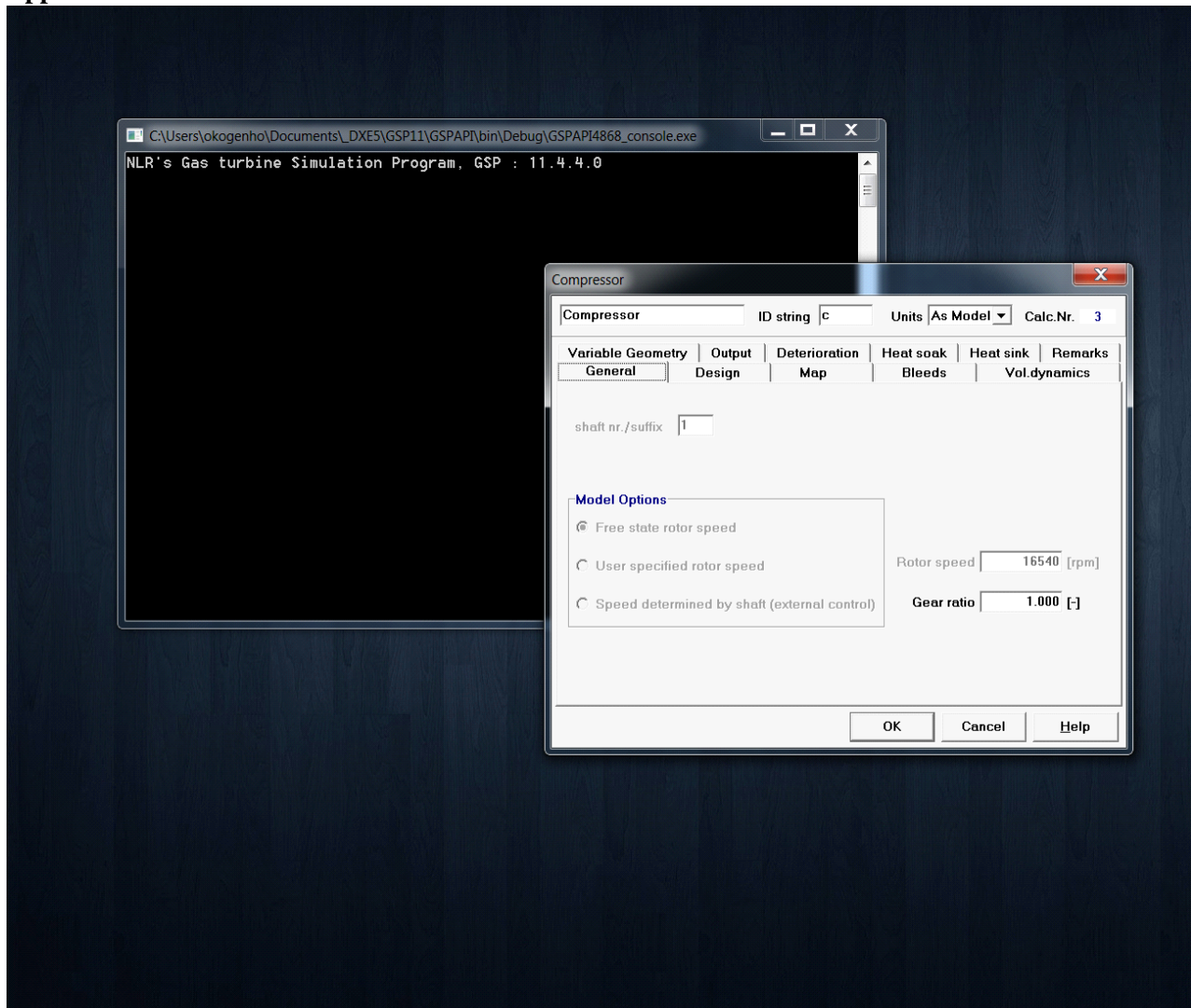
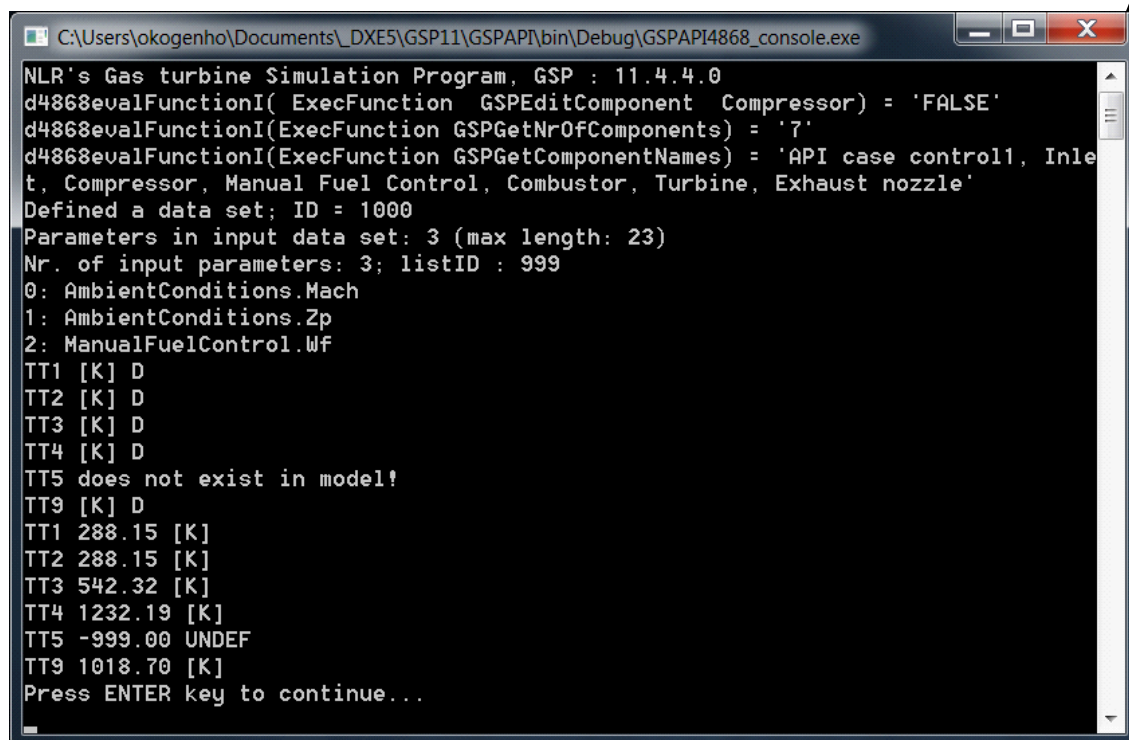


Figure 9 – Desktop screenshot using GSP API in Delphi console application during execution



```

C:\Users\okogenho\Documents\DXE5\GSP11\GSPAPI\bin\Debug\GSPAPI4868_console.exe
NLR's Gas turbine Simulation Program, GSP : 11.4.4.0
d4868evalFunctionI( ExecFunction GSPeditComponent Compressor) = 'FALSE'
d4868evalFunctionI(ExecFunction GSPGetNrOfComponents) = '7'
d4868evalFunctionI(ExecFunction GSPGetComponentNames) = 'API case control1, Inlet, Compressor, Manual Fuel Control, Combustor, Turbine, Exhaust nozzle'
Defined a data set; ID = 1000
Parameters in input data set: 3 (max length: 23)
Nr. of input parameters: 3; listID : 999
0: AmbientConditions.Mach
1: AmbientConditions.Zp
2: ManualFuelControl.Wf
TT1 [K] D
TT2 [K] D
TT3 [K] D
TT4 [K] D
TT5 does not exist in model!
TT9 [K] D
TT1 288.15 [K]
TT2 288.15 [K]
TT3 542.32 [K]
TT4 1232.19 [K]
TT5 -999.00 UNDEF
TT9 1018.70 [K]
Press ENTER key to continue...

```

Figure 10 – Delphi console application screenshot at finish of application run

5.4 Error and warning messages

The error- and warning messages are listed in the sections called [List of errors](#) and [List of warnings](#).

5.4.1 List of errors

The following list shows the errors the GSP API may display to the GSP API user/customer during execution.

Table 17 – List of error messages

Error	Message	Remark
-101, -201, -301, -401, -501, -502, -503, -601, -701, -801, -901, -1001, -1101, - 1201, - -1301, - 1501, - -1601, - 1701, - -1801, - 1901, - -2001, - 2101, - -2201, -2401	'Function "%s" called prior to function "d4868initProg!"'	%s will be replaced by the function name from which the error has been called



Error and warning messages

-102	'Log file cannot be opened for writing!'	-
-302	'Function "%s" called with incorrect data input type "%s"'	%s will be populated with relevant data
-303	'Function "%s" called while no model loaded!'	%s will be replaced by the function name from which the error has been called
-402	'Function %s called for a scalar parameter!'	%s will be replaced by the function name from which the error has been called
-1803	'Too few string elements in string to execute command in string to parse!'	-
-1804	'Unsupported function call!'	-
-1804	'Unsupported procedure call!'	-
-1805	'Unrecognised execute command in string to parse!'	-
-2002	'Cannot set data for current listID (%d: listID 999 can only be used for setting data!'	%d will be populated with relevant data
-2003	'No API input interface component found in the model!'	-
-2102	'No API input interface component found in the model!'	-
-2402	'Failed to free GSP API allocated memory!'	-
-97201	'-a.201: Function "d4868loadModel" called prior to function "d4868initProg"'	-
-97301	'-a.301: Function "d4868closeModel" called prior to function "d4868initProg"'	-
-97302	'-a.302: Function "d4868closeModel" called prior to function "d4868loadModel"'	-
-97401	'-a.401: Function "d4868saveModel" called prior to function "d4868initProg"'	-
-97402	'-a.402: Function "d4868saveModel" called prior to function "d4868loadModel"'	-
-97501	'-a.501: Function "d4868showModel" called prior to function "d4868initProg"'	-
-97502	'-a.502: Function "d4868showModel" called prior to function "d4868loadModel"'	-
-97601	'-a.601: Function "d4868getInputDataList" called prior to function "d4868initProg"'	-
-97602	'-a.602: No API input interface component found in the model!'	-
-97603	'-a.603: Not enough memory reserved (%d for storing the input parameters (%d)!'	%d will be populated with relevant data

-97701	'-a.701: Function "d4868getInputDataListSize" called prior to function "d4868initProg"!'	-
-97702	'-a.702: No API input interface component found in the model!'	-
-97801	'-a.801: Function "d4868programInfo" called prior to function "d4868initProg"!'	-
-97901	'-a.901: Function "%s" called prior to function "d4868initProg"!'	%s will be replaced by the function name from which the error has been called
-97902	'-a.902: No function result returned!'	-

5.4.2 List of warnings

The following list shows the warnings the GSP API may display to the GSP API user/customer during execution.

Table 18 – List of warning messages

Warning	Message	Remark
101	'Log file already exists, file will be overwritten!'	-
201	'No log file opened -> cannot close log file!'	-
401, 901, 1001, 1101, 1601, 1701, 2101, 2201, 2202, 2203, 2301, 2302, 2303, 2311, 2312, 2313, 2321, 2322, 2323, 2331, 2332	'Function "%s" is currently an unsupported function!'	%s will be replaced by the function name from which the error has been called.
601	'Note that all output parameters in GSP are stored in double-precision floating-point format!'	-
602	'dataType string (%d has been clipped to fit buffSize (%d)!'	%d will be replaced by actual values of the parameter sizes
701	'In GSP no description property is defined, function results an empty string!'	-
1201	'In GSP no severity index error handling is implemented, 999 (unknown) is returned!'	-
1401	'Trying to call "d4868initProg" again is not allowed: function execution skipped!'	-
97901	'a.901: Allocated string length (%d) too short (%d)!'	%d will be populated with relevant data



GSP

VI

6 Registration & Support

6.1 Contact details

For questions, problems or support please contact NLR using the special feedback forms on the GSP site at www.gspteam.com or directly through info@gspteam.com. Registered customers can contact the GSP team using the support@gspteam.com e-mail address.

6.2 Registration

GSP is protected with a registration key. An unregistered version will have limited capabilities, i.e. saving of many non-standard components. Registration of GSP gives you the benefit of NLR support in the form of answering questions, assisting in solving problems and development of new models according to the license agreement.

For license and registration information, please navigate to the [licensing](#) and [order](#) web pages. If there are further questions please do not hesitate to [contact us](#).

The registration code is coupled to a specific registrant name. The registration information is supplied through e-mail upon payment delivery. Copy both the registrant name and the registration key into the input boxes of the registration window.

6.2.1 Registration window

Open the registration window by selecting the Registration menu item from the 'GSP main window' Help menu.

Copy the registrants name supplied in the registration e-mail into the Registration name input field and copy the registration key into the Registration code input field. "Copy-paste" is preferred above typing the name and code to prevent typing errors (e.g. zeros, 0, and capital letters O look very similar). Press OK to register the application.



6.3 Support from NLR

The National Aerospace Laboratory NLR only supports [registered](#) users of GSP by answering questions, assisting in solving problems, development of new models or development of customized components, according to the limitations in the applicable license agreement. Optional support can be purchased.



Since GSP is continuously being extended and updated following user suggestions, we appreciate comments and bug reports of both users of the standard version as well as registered users.

When contacting for a support question, please report

- the GSP version number
The version can be found by clicking `Help | About` in the main program window, or alternately right-clicking the file icon of the `GSP.exe` file typically located in the installation directory, e.g. `.\Program Files\NLR\GSP\`, selecting `Properties` and looking in the `Version` tab sheet), and
- describe the problem, or systematically describe how the problem arises
- if possible do send the model/project so that is is easier for the GSP development team to reproduce the error

A list of Frequently Asked Questions can be found on the GSP home site at www.gspteam.com.



GSP

VII

References

1. Visser W.P.J., Broomhead M.J., 2000, **GSP, A Generic Object-Oriented Gas Turbine Simulation Environment**, ASME 2000-GT-0002, ASME IGTI Turbo Expo conference, Munich, 2000
2. Tinga T., Visser W.P.J., de Wolf W.B., and Broomhead M.J., 2000, **Integrated Lifing Analysis Tool for Gas Turbine Components**, ASME 2000-GT-0646, ASME IGTI Turbo Expo conference, Munich 2000
3. Visser W.P.J., 1995, **Gas Turbine Simulation at NLR, Making it REAL**, CEAS Symposium on Simulation Technology (paper MOD05), Delft, the Netherlands
4. Visser W.P.J., Kluiters S.C.M., 1999, **Modeling the Effects of Operating Conditions and Alternative Fuels on Gas Turbine Performance and Emissions**, NLR Technical Publication NLR-TP-98629 or Research and Technology Organization RTO-MP-14
5. Visser W.P.J., Kogenhop O., Oostveen. M., 2004, **A Generic Approach for Gas Turbine Adaptive Modeling**, ASME GT2004-53721, ASME IGTI Turbo Expo conference, Vienna 2004 and ASME Journal of Engineering for Gas Turbines and Power, Volume 128 2006
6. Visser W.P.J., Oostveen M., Pieters H., Dorp E. van, 2006 **Experience with GSP as a Gas Path Analysis Tool**, ASME GT2006-90904, ASME IGTI Turbo Expo conference, Barcelona 2006
7. Michel L. Verbist, Wilfried P.J. Visser, Jos P. van Buijtenen, Rob Duivis, **Gas Path Analysis on KLM in-Flight Engine Data**, ASME paper GT2011-466257, presented at the ASME TURBO EXPO 2011, 6-10 June, Vancouver, Canada
8. Verbist, M.L., Visser, W.P.J., van Buijtenen, J.P., Duivis, **Model-based gas turbine diagnostics at KLM Engine Services**, XX ISABE Conference, 2011, Gothenburg, Sweden, ISABE-2011-1807
9. Verbist, M.L., Visser, W.P.J., Pecnik, R., van Buijtenen, J.P., **Component Map Tuning Procedure using Adaptive Modeling**. ASME Turbo Expo 2012, Copenhagen, Denmark, GT2012-69688
10. **Manual of the ICAO standard atmosphere extended to 80 kilometres (262 500 feet)**, ICAO 7488/3, 1993
11. Kurzke J., 1996, **How to get Component Maps for Aircraft Gas Turbine Performance Calculations**, ASME 96-GT-164
12. Kurzke J., 2005, **How to Create a Performance Model of a Gas Turbine From a Limited Amount of Information**, ASME GT 2005-68537
13. RTO TECHNICAL REPORT RTO-TR-044
"Performance Prediction and Simulation of Gas Turbine Engine Operation"
14. RTO TECHNICAL REPORT TR-AVT-036

Index

- A -

authors 11

- B -

bug fixes 10

- C -

compiler 17
contact NLR 76

- D -

documentation 10

- F -

fixes 10
forum 76

- G -

GSPteam 11

- I -

improvements 10

- N -

new features 10

- P -

programmers 11

- R -

references 79
registration 76
release notes 10

- S -

support 76
system requirements 9

- W -

website 76
welcome 5
what's new? 10
writers 11



www.GSPteam.com



Download GSP