



AEROSPACE RECOMMENDED PRACTICE

ARP7998™

Issued

2023-03

Shared Memory Interface for Gas Turbine Engine Performance Programs

RATIONALE

Shared memory (SHM) is a common methodology of computer inter-process communication (IPC) to interface between two separate executables running in their own process or address space. This SAE Aerospace Recommended Practice (ARP) proposes a general approach for creating a standard interface and interactions to allow this type of interface to be easily utilized in gas turbine engine simulations for real-time model execution, standardized third party interfaces or multiple discipline model executions in the same computing space.

TABLE OF CONTENTS

1.	SCOPE.....	3
2.	REFERENCES.....	3
2.1	Applicable Documents	3
2.1.1	SAE Publications.....	3
2.2	Definitions	3
3.	OVERVIEW	4
4.	SHARED MEMORY ALLOCATION	4
4.1	Data Structure	4
4.1.1	Recommended C Style Structure Type Block for Minimum Shared Memory Block.....	5
4.2	Block Variable Definitions	5
4.3	Shared Memory Initialization.....	6
4.3.1	Windows® VC++ Example.....	6
4.3.2	Linux Methods.....	6
4.4	Semaphore Setup	7
4.4.1	Windows® Example.....	8
4.4.2	Linux Example.....	8
5.	BLOCK INTERACTION DEFINITIONS.....	9
5.1	Interactions Examples.....	11
5.1.1	Set Inputs Example	11
6.	NOTES	12
6.1	Revision Indicator.....	12

SAE Executive Standards Committee Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be revised, reaffirmed, stabilized, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2023 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: Tel: 877-606-7323 (inside USA and Canada)
Tel: +1 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
http://www.sae.org

SAE WEB ADDRESS:

Full document details: SAE ARP7998

For more information on this standard, visit
<https://www.sae.org/standards/content/ARP7998/>

Documents

Document Sharing Hub

APPENDIX A

WINDOWS® C++ SERVER EXAMPLE

13

APPENDIX B

WINDOWS® C++ CLIENT EXAMPLE

19

APPENDIX C

LINUX C++ SERVER EXAMPLE

23

APPENDIX D

LINUX C++ CLIENT EXAMPLE

28

1. SCOPE

This is an initial release of a recommended practice for an interface to supplier simulations that utilize traditional inter-process communication (IPC) methods of shared memory and semaphore communications. These IPC methods are standard practice in the computer science world and allow for communication by separate processes running on a computer without any common runtime requirements. This allows things like 32-bit applications to talk with 64-bit applications, as well as any other compiler version or runtime library dependencies being required by the calling program to interface with the called model. This also allows the calling program and the called program to be run on separate CPUs to allow parallel execution of the called program as well as multiple instances of the called program to execute all on separate processors. Also, other third party applications would be able to interact with this common IPC interface allowing an interface where no compilation or integration of either parties libraries or code is required. This SAE Aerospace Recommended Practice (ARP) provides definition of the shared memory space and interactive IPC functions to enable the interaction between these separate applications.

2. REFERENCES

2.1 Applicable Documents

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

2.1.1 SAE Publications

Available from SAE International, 400 Commonwealth Drive, Warrendale, PA 15096-0001, Tel: 877-606-7323 (inside USA and Canada) or +1 724-776-4970 (outside USA), www.sae.org.

ARP4868	Function-Based API for Gas Turbine Engine Performance Programs
ARP5571	Gas Turbine Engine Performance Presentation and Nomenclature for Object-Oriented Computer Programs
AS681	Gas Turbine Engine Performance Presentation for Computer Programs
AS4191	Gas Turbine Engine Performance Presentation for Computer Programs Using Fortran
AS6502	Aircraft Propulsion System Performance Nomenclature

2.2 Definitions

API: Application program interface, a protocol agreed to between the supplier and the customer for interaction between the customer's computer program and the supplier's engine performance program, when the supplier's program is operating as a subprogram.

CALLING PROGRAM: The program responsible for interfacing with the engine program through the interface methods defined in this document.

CUSTOMER: The person or organization which receives and executes the engine program.

ENGINE PROGRAM: Gas turbine engine performance program. The API functions defined in this document are supplied as a part of the engine program.

INTER PROCESS COMMUNICATION (IPC): In computer science, interprocess communication (IPC) refers specifically to the mechanisms an operating system provides to allow processes it manages to share data.

PARAMETER NAME: ENGINE PROGRAM variable names as defined in the engine program documentation.

SEMAPHORE: A semaphore is a synchronization object that controls access by multiple processes to a common resource in a parallel programming environment. Semaphores are widely used to control access to files and shared memory. The three basic functionalities associated with semaphores are set, check and wait until it clears to set it again

SHARED MEMORY (SHM): In computer science, shared memory is a single computer memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies. Shared memory is an efficient means of passing data between programs.

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

VC/MSVC: Reference to the Windows® Microsoft Visual Studio C/C++ Compiler ([MSVC](#)) that sometimes is an issue on integrating a version of the supplier model as it is tied to the C++ Standard version itself, as it has been adjusted over the years to remain competitive with the current programming languages of today.

3. OVERVIEW

Historically gas turbine engine programs have used a Fortran memory structures that relied on common blocks for data communication. This was predicated on the customer either working totally in the supplier code environment or adapting it to run as a subprogram to an application that still interacted within the Fortran environment. Most customers still wrap interfaces around supplier models, and their higher-level analysis codes are setup to treat the supplier models like a black box. However, most suppliers have moved away from Fortran based systems in favor of object-oriented paradigms. So legacy customer systems have needed to adapt to utilize API interfaces like an ARP4868 to interface to a model. These function- based interfaces are not as efficient as the legacy Fortran common block approach and can hinder real-time execution speeds for data exchange and also puts requirements to match compiler and runtime library versions, 32/64-bit limitations on the interface. Shared memory is a lot like legacy Fortran common blocks and offers a method of exchanging information that is both speed efficient as well as removing any compiler and runtime requirements from the customer host program. For example, on the Microsoft Windows® platform a Visual Studio® C/C++ (VC14) 64-bit code can interact with an older VC10 32-bit application by just interacting with the common shared memory space. Other benefits of this type of interface is that the supplier model runs under its own process, this can be dedicated to its own CPU and prioritized for maximum execution speed. Also, if multiple instances of the supplier model are needed, either multiple engines or parallel model execution, each will run on its own CPU and interact with the customer host with no speed impacts. These methods and processes have been available and utilized for many years on both the Windows® and Linux/Unix environments with this ARP being an effort to help facilitate this type of interface.

The intent of this ARP is to provide guidelines for setup of a shared memory block, API calls to interact with memory, and methods for defining unique semaphore names as well as interactions between the calling program and the supplier model. Semaphores are a common computer science IPC practice to signal that changes have occurred in one process that require another process to do something. A recommend process for setting up these semaphores along with providing information to the calling program on what to do when called will be defined in this ARP.

4. SHARED MEMORY ALLOCATION

4.1 Data Structure

As stated before, shared memory blocks are similar to Fortran common blocks; however, the memory space is outside the executing program. Most shared memory allocations utilize a C struct to setup the memory block. Below is a recommend minimum for a typical interface. This provides space for input and output arrays along with space for other recommended setting flags to communicate between programs.

4.1.1 Recommended C Style Structure Type Block for Minimum Shared Memory Block

```
typedef struct          /* ARP Shared Memory Block Structure */
{
    int    IOFLAG;       // Signals what you want the calling program to do with the SHM Contents
    int    ERRFLAG;      // Error Status of the last operation done by the Supplier code
    int    SIMMODE;      // Execution mode of the model 1 = Steady State 2= Save Guess 3= Transient
    int    DEBUG;        // Flag to enable more verbose messaging 1=on 0=off (default)
    int    NumInputs;     // Number of Inputs Mapped through the interface
    int    NumOutputs;    // Number of Outputs Mapped through the interface
    char    COMMAND[1024]; // String Slot used to pass names and Commands through the interface
    long    FRAMECOUNT;  // Counter for each execution frame
    char    ERRMSG[1024];  // String Slot used to pass error message associated with ERRFLAG
    double  TIME;         // Transient time the simulation was at for the last update
    double  INPUTS[1000];  // Array of input data values matched to the name array order
    double  OUTPUTS[2000]; //Array of output data values matched to output name array order
} ENGINE_SHM_Type;
```

This minimal block provides enough storage space and communication flags to easily interface to any simulation. It keeps two arrays on input and output variables. Note size just allocates memory for the arrays; the number of inputs and number of outputs tells the calling program how many variables are actually being passed through the interface. The list of inputs and outputs can be specified in the supplier simulation by a supplied input that is customizable by customer or defined by the customer at runtime from the client side of the interface.

4.2 Block Variable Definitions

IOFLAG: Signal to the supplier model instruct it what to do with the SHM contents. See Section 5 for details.

ERRFLAG: Returns any error code in the SHM interface or from model execution.

SIMMODE: Sets the run mode of the mode (1= steady state, 2 = steady state save guess, 3 = transient).

DEBUG: Turns on verbose messagin in the interface for debugging and diagnostic purposes.

NumInputs: This gets set when the engine model loads to define the number of the input variables being mapped through interface or can be set by the calling program. Note the INPUTS[] array is defaulted to 1000 variables; however, this variable tells the calling program how many are actually assigned.

NumOuputs: This gets set when the engine model loads to define the number of the output variables being mapped through interface, but can be defined by the calling program as well. Note the OUTPUTS[] array is defaulted to 2000 variables; however, this variable tells the calling program how many are actually assigned.

COMMAND[]: This is a dedicated character block that can be used for passing a string command to be executed in the supplier model, also for passing other string items like variable names out of the supplier model.

FRAMECOUNT: This is a long integer incremented each time a command is executed in the SHM interface, used for timing and data synchronization.

TIME: This is the simulation time that matches the outputs (if transient data).

INPUTS[]: Values of input variables for the current run.

OUTPUTS[]: Returned values from the supplier model.

4.3 Shared Memory Initialization

With the block defined, the memory block needs to be initialized and made available. This is done by the supplier portion of the code when it is executed. The recommended setting below utilizes the process ID as part of the shared memory block name so that each instance of supplier code has unique memory space. This allows multiple engines to be run at the same time without interfering with each others I/O. Process ID numbers are unique to each executing process on a machine and are not duplicated. Below is a standard C++ implementation for a Windows® version of the shared memory initialization.

4.3.1 Windows® VC++ Example

```
typedef struct {
void *addr; char *shm_file_name; HANDLE shm_file_handle;
} engine_shm_t; // An additional block to store info about the SHM Block

ENGINE_SHM_Type* SHMBlk; // Pointer to SHMBlock
engine_shm_t data; // Instance of the SHM memory space defined in section 4.1

stringstream file_name; // Maps the process ID to a fixed string for a unique block name
long PID = _getpid();
file_name << "NPSS_SHM_FILE_MAP." << PID;
string Fname = file_name.str();

HANDLE file_handle = CreateFileMapping(INVALID_HANDLE_VALUE, // system pagefile
NULL, // optional security attributes
PAGE_READWRITE, // protection for mapping object
0, // high-order 32 bits of object size
sizeof(FM_ENGINE_SHM_Type), // Size of Block
(LPCSTR)file_name.str().c_str()); //name of file - mapping

data.shm_file_handle =file_handle;
data.shm_file_name = (char*)Fname.c_str();
data.addr = MapViewOfFile(data.shm_file_handle, FILE_MAP_WRITE,0,0,sizeof(FM_ENGINE_SHM_Type));

SHMBlk = (ENGINE_SHM_Type*)data.addr; // Point the local pointer to the SHM block

SHMBlk->IOFLAG = 1; // Now we can set members of the Shared Memory Space
```

4.3.2 Linux Methods

Linux methods are taken from the older Unix operating systems and are different than Windows® but provide the same functionality. The shared memory block definition is the same as defined in 4.1. However differences are shown below. Again, it is recommended that the server executable setup all of the shared memory and semaphores.

```
typedef struct {
int shm_id;
key_t key;
} engine_shm_t; // Struct type to store info about the SHM Block

/* SET UP SHARED MEMORY BLOCK */
ENGINE_SHM_Type* SHMBlk; // Pointer to SHMBlock
engine_shm_t data; // Instance of the memory space

data.key=getpid(); // We use this to create unique tags for the ShM and Sems

// Create Shared Memory Block named after the process ID
data.shm_id = shmget( data.key, sizeof(ENGINE_SHM_Type), IPC_CREAT | SHM_W | SHM_R | (SHM_R >> 6) );

/* Now Attach our local instance to it*/
SHMBlk = (ENGINE_SHM_Type *) shmat( data.shm_id, NULL, 0 );

SHMBlk->IOFLAG = 1; // Now we can set members of the Shared Space
```

On the client side, typically the client will either spawn the server on demand or hook into an existing process. So it just needs to wait till the shared memory block exists and set things up after that. With the server side already in place, the client could hook in like the following:

```
// This waits until the ShM Block Exists in the server, it will go positive
while(data.shm_id ==-1){
    data.shm_id = shmget( data.key, sizeof(ENGINE_SHM_Type),0666 );
}

/* Now Attach the Shm to the local instance */
if ( data.shm_id ) {
    SHMBlk = (ENGINE_SHM_Type *) shmat( data.shm_id, NULL, 0 );
    if ( SHMBlk == (void*) -1 ) {
        cout <<" Error Attaching Shared Memory in Client"<<endl;
    }else {
        printf( "testClient-> PropIO Attached to shmid: %d\n", data.shm_id );
    }
}

}
```

Now both the client and server can communicate through the unique identifier.

4.4 Semaphore Setup

Semaphores are an IPC signaling mechanism that allow different processes to synchronize their access to certain resources. They are typically used with shared memory applications where two applications need to signal each other to take action on the data in the SHM block. Semaphores are also useful for putting an executable in a wait state, while the customer host program is doing other things, the supplier model can sit in a non-executing state awaiting the signal to take action. This ARP recommends that only two are required to enable communication between the supplier and customer interface. One to signal the that the customer calling program has sent information to the shared memory block and it needs to process the request and another for the supplier model to signal it has completed the request. Also recommended, as with the shared memory block, a unique identifier is needed so that multiple simulation instances do not have any mixed signals. Appending the process ID to the semaphore name is used to prevent this interaction.

Recommend 2 System Semaphore Names:

Data Sent: DS_Flag.PID#

Run Complete: RC_Flag.PID#

Below is an example C++ Windows® recommended setup for semaphores.

4.4.1 Windows® Example

```
// Initialize the Semaphores
HANDLE DS_Semaphore = NULL; // Data Sent Semaphore
HANDLE RC_Semaphore = NULL; // Run Complete Semaphore
LONG cMax = 1000;
LPSTR DS_Sem;
LPSTR RC_Sem;
DWORD dwWaitResult; //Wait parm
LONG pOldVal = 0; // Used for passing Arg

// this names the semaphores based on the process ID they were created under
char buffer[32];
wsprintf(buffer, "%d", InID);

string DS(buffer);
string temp = "DS_Flag." + DS;
DS_Sem = (LPSTR)temp.c_str();
string temp2 = "RC_Flag." + DS;
RC_Sem = (LPSTR)temp2.c_str();

// This sets up the Data Sent Semaphore
DS_Semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS, TRUE, DS_Sem);
if (DS_Semaphore == NULL) {
    cout << "%CLIENT%-> ERROR connecting to Data Sent Semaphore\n";
}
// This sets up the Run Complete Semaphore need to have this after the client launches
RC_Semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS, TRUE, RC_Sem);
if (RC_Semaphore == NULL) {
    cout << "%CLIENT%-> ERROR connecting Run Complete semaphore\n";
}
Sleep(50);
SHMBlk->IOFLAG = 0;
ReleaseSemaphore(DS_Semaphore, 1, &pOldVal); // Signal Engine to be Ready
dwWaitResult = WaitForSingleObject(RC_Semaphore, INFINITE); // Wait till its Ready
```

4.4.2 Linux Example

Linux methods are based on older Unix OS commands but are of similar functionality. A recommended Linux semaphore setup is shown below:

```
/*Set up semaphores */
string DS_NAME ="DS_Flag."
string RC_NAME ="RC_Flag."

sem_t *RC_Semaphore;
sem_t *DS_Semaphore;

stringstream ss;
ss<<data.key; // Use the process ID from above

DS_NAME.append(ss.str());
RC_NAME.append(ss.str());

/*The Semaphores are created in the Server Executable */
RC_Semaphore = sem_open(RC_NAME.c_str(),O_CREAT,0777,0); // Create the RC Flag
DS_Semaphore = sem_open(DS_NAME.c_str(),O_CREAT,0777,0); // Create the DS Flag
```

After creation, the client executable would connect using the same definitions but using the following commands:

```
// Hooking up to the DS and RC flags in the server process from a client
```

```
DS_Semaphore = sem_open(DS_NAME.c_str(),0); //Set the Data Sent Semaphore created from the server
```

```
RC_Semaphore = sem_open(RC_NAME.c_str(), 0); // Set the Run Complete Semaphore created from the server
```

Once the semaphores are attached in the client/server, they can be signaled using the appropriate methods.

For example, in the client:

```
sem_post(DS_Semaphore); //Set the Data Sent flag
sem_wait(RC_Semaphore); //Wait till the servers says its done.
```

5. BLOCK INTERACTION DEFINITIONS

With the shared memory block defined and semaphores in place, it is now time to define the transactions that will occur through the block between the customer and supplier models. This ARP recommends using a single flag variable IOFLAG that exists in the block to signal what needs to be done. By setting the IOFLAG to specific value the supplier code will know what it needs to do when the data sent semaphore signals it. This allows several transactions to be allowed to pass data back and forth between the customer and supplier processes. Below is a recommended list of transactions that correspond to various settings of the IOFLAG. Various ranges have been selected to group functions of similar tasks.

The IOFLAG has been grouped in sets of ten based on its function:

- ≤0 through 4: Reserved for internal communication
- 5: Model load command
- 10 through 19: Set/get input names
- 20 through 29: Set/get output names
- 30 through 39: Input value set options
- 40 through 49: Run/execute model options
- 50 through 59: Output value get options
- 60 through 99: Miscellaneous interactions
- 70 through 89: Open for user
- 90 through 99: Error/debug options

IOFLAG = 5 Model load and initialize command.

This uses the string value in the COMMAND variable to load and initialize the engine model. This is optional and only required if the executable command did not start the model already. Similar to the 4868 init command. Individual arguments are fed in through by setting the COMMAND variable.

IOFLAG = 10 Get available input variable names one by one through the COMMAND variable in the block.

This contains a loop feature that allows the client to get a list of input variables already defined in the SHM Block be provided back to the calling program. This gives an ordered list that matches up to the INPUTS[] array value order.

- IOFLAG = 12** Set input variables names one by one through the COMMAND variable in the block.
- This will set the Input list of variables from the client program, overriding any predefined list in the server. This provides a new list of inputs to be used.
- IOFLAG = 14** Append input variables names one by one through the COMMAND variable in the block.
- This will append to the already existing list of input variables in the block. The new input will be appended to the current stack.
- IOFLAG = 16** Remove input variables names one by one through the COMMAND variable in the block.
- This will remove any existing Input list entry from the block and move all of the existing entries up the stack.
- IOFLAG = 20** Get available output variable names one by one through the COMMAND variable in the block.
- This contains a loop feature that allows the list of output variables already defined in the SHM block to be provided back to the calling program. This gives an ordered list that matches up to the OUTPUTS[] array value order.
- IOFLAG = 22** Set available output variable names one by one through the COMMAND variable in the block.
- IOFLAG = 24** Append output variables names one by one through the COMMAND variable in the block.
- This will append to the already existing list of output variables in the block. The new output will be appended to the current stack.
- IOFLAG = 26** Remove output variables name in the COMMAND variable in the block from the output stack.
- This will remove any existing output list entry from the block and move all of the existing entries up the stack.
- IOFLAG = 30** Signals the server from the client program that the INPUTS[] values have been set by the calling program, now the server shall set the values in the supplier model.
- IOFLAG = 32** Set the input defined in the COMMAND variable to the value of INPUTS[0].
- This allows a single set of a defined variable in the COMMAND slot to the value in the INPUTS[0] slot. This is useful for setting variables not defined in the input array list.
- IOFLAG = 40** Run the supplier model.
- This tells the supplier model to execute the model.
- IOFLAG = 42** Parse the string value of the COMMAND variable.
- This utilizes a parse string functionality that allows the calling program so set string variables or issue other commands in the supplier model.
- IOFLAG = 44** Override the model run() command with the value in the COMMAND variable.
- This allows another function to be used as the run command in the supplier model, an NPSS example would be to override the run(); command with myRun();. This command is what gets executed in IOFLAG=44 which defaults to run();.