



---

## **Multi-Tasking Features for PathMATE Transformation Maps**

Version 3.0  
November 7, 2007

---

### *PathMATE Technical Notes*

Pathfinder Solutions LLC  
33 Commercial Drive, Suite 2  
Foxboro, MA 02035 USA  
[www.PathfinderMDA.com](http://www.PathfinderMDA.com)  
888-662-7284

# Table Of Contents

<b>1. Introduction.....</b>	<b>1</b>
<b>2. Philosophy .....</b>	<b>1</b>
Task Proliferation in Traditional Systems .....	1
The Event-Driven Paradigm .....	1
<b>3. Design Approach.....</b>	<b>2</b>
<b>4. Capabilities .....</b>	<b>4</b>
Domain Allocation .....	4
<i>Analysis Property Extensions .....</i>	<i>5</i>
<i>Translation Features .....</i>	<i>6</i>
<i>Mechanisms Features .....</i>	<i>6</i>
<i>Analysis Implications .....</i>	<i>6</i>
Domain Service Allocation .....	7
<i>Analysis Property Extensions .....</i>	<i>8</i>
<i>Translation Features .....</i>	<i>9</i>
<i>Mechanisms Features .....</i>	<i>9</i>
<i>Analysis Implications .....</i>	<i>9</i>
Class Allocation.....	10
<i>Analysis Property Features .....</i>	<i>11</i>
<i>Translation Extensions Features .....</i>	<i>12</i>
<i>Mechanisms Extensions.....</i>	<i>12</i>
<i>Analysis Implications .....</i>	<i>12</i>
Dynamic Domain Service Allocation .....	13

## 1. Introduction

This Technical Note describes the capabilities to be deployed in support of multi-tasking for the PathMATE Transformation Maps. These capabilities are compatible with multi-process and multi-processor capabilities as specified in the Distributed Deployment Technical Note.

## 2. Philosophy

### Task Proliferation in Traditional Systems

Traditional embedded systems development calls on specific skill sets that are needed to deploy complex functionality in constrained execution environments. Prominent among these skills is the ability for an embedded systems programmer to break up their system into separate threads of control – tasks – and craft their interaction. This is done to achieve apparent parallelism and to simplify the programming of each functional element using primarily synchronous programming languages (referring to function calls in C, etc.).

The coincidence of a rapid increase in complexity in embedded systems and common support for multi-tasking in most embedded operating systems have resulted in a proliferation of tasks in embedded systems. The incremental growth of established systems also promotes the creation of new tasks to conveniently add new capabilities to existing implementation architectures. Overall, hand-programmed embedded systems tend to use a larger number of tasks than is required, incurring unnecessary run-time overhead in context switching and contention mechanisms, and needlessly complicating debug and maintenance.

### The Event-Driven Paradigm

The use of MDA models breaks some of the constraints of the hand-programming approach. With UML statecharts capturing class lifecycles, a choice is available to the synchronous function call. In addition to the explicit addition of an asynchronous choice in the UML Event, an implicit choice is also now available. By using the flexibility of template-based translation, even synchronous service calls can be implemented as asynchronous messages in some cases. Both of these implementation options support the allocation of much larger fragments of processing to a single RTOS task, allowing the event queue mechanism to thread together the activities of many state machines and the dispatch of many asynchronous service calls via service handles – all in a single RTOS task. So the use of MDA/translation reduces the number of tasks needed to effectively deploy a system.

While these benefits of MDA/translation can dramatically reduce the need for many separate RTOS tasks at the implementation level, in MDA systems there generally remains a need for two or more tasks. In addition to multiple tasks running code generated from analysis, there can be one or more tasks with realized (hand written) code from legacy or other sources.

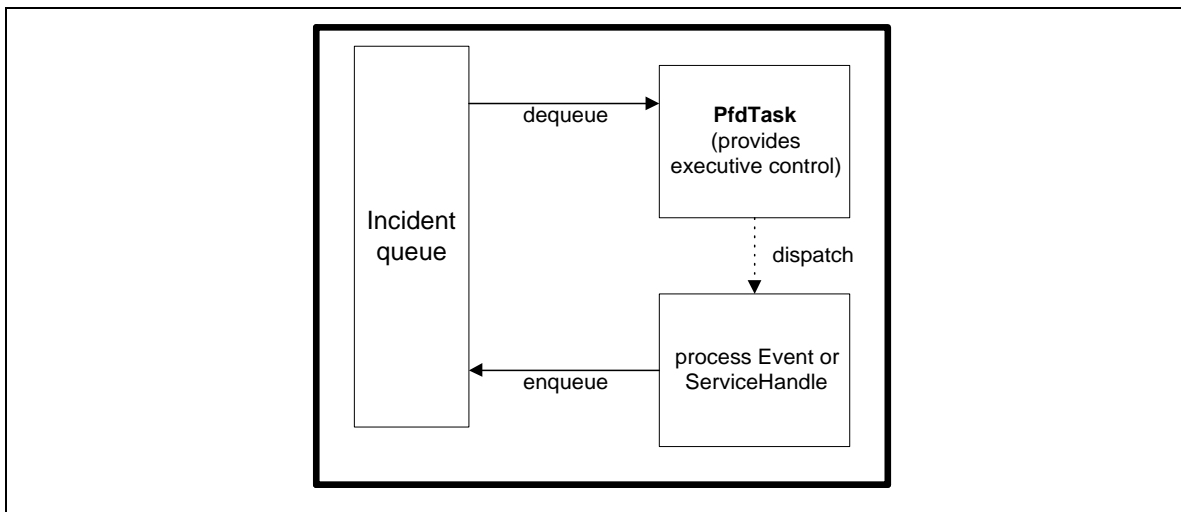
So where does this leave us? We have eliminated the freewheeling tendency to proliferate tasks. So our need for inter-task synchronization mechanisms

is greatly reduced, and our run-time overhead is reduced. But often more than one task is needed, even with the implementation opportunities afforded by MDA and translation. What remains is the requirement to deploy relatively large-scale system components to a small number of tasks. Pathfinder's strategy is to focus on the domain as the basis of allocation.

The design alternatives described herein have a goal of providing the a level of tasking flexibility while minimizing task locking overhead/requirements, and preserving the implementation independence of the analysis models.

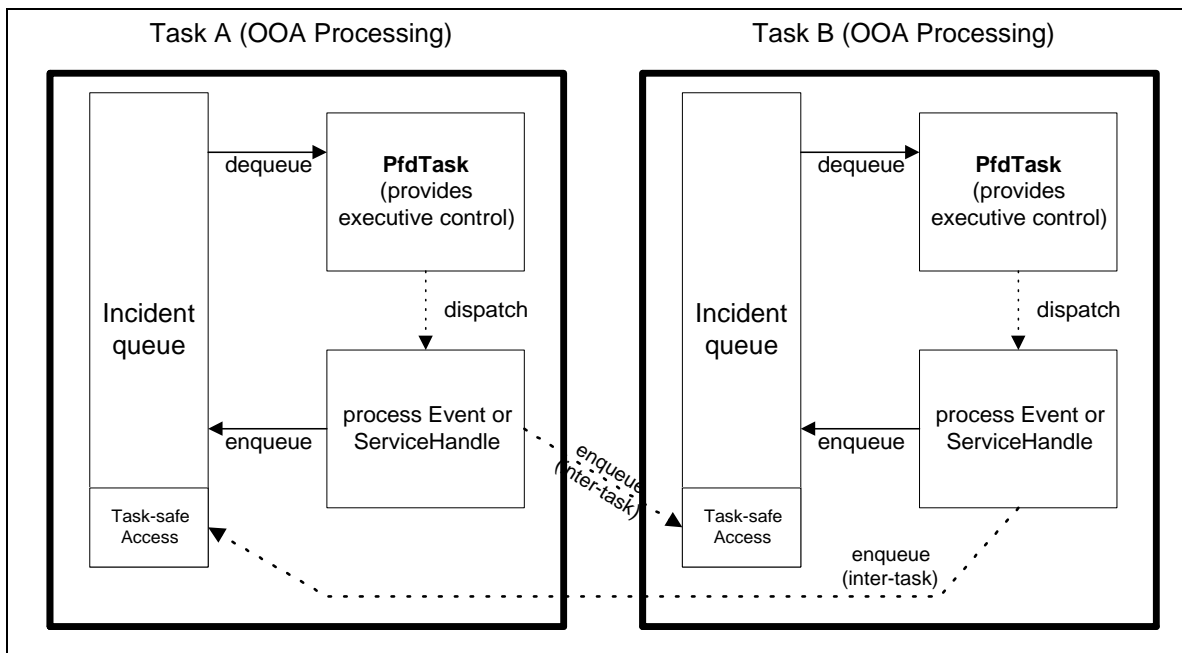
### 3. Design Approach

The execution of generated implementation from MDA models – called "OOA Processing" - revolves around the central execution control mechanism PfdTask (SW\_Task in the C design). This contains Incident queues, regulating the dispatch of Events and ServiceHandles in the course of OOA Processing. Everything that happens in an OOA Processing task is a response to an Incident - Event or ServiceHandle. Of course these impulses causes many other things to happen, but from an implementation architecture perspective everything is driven from them.



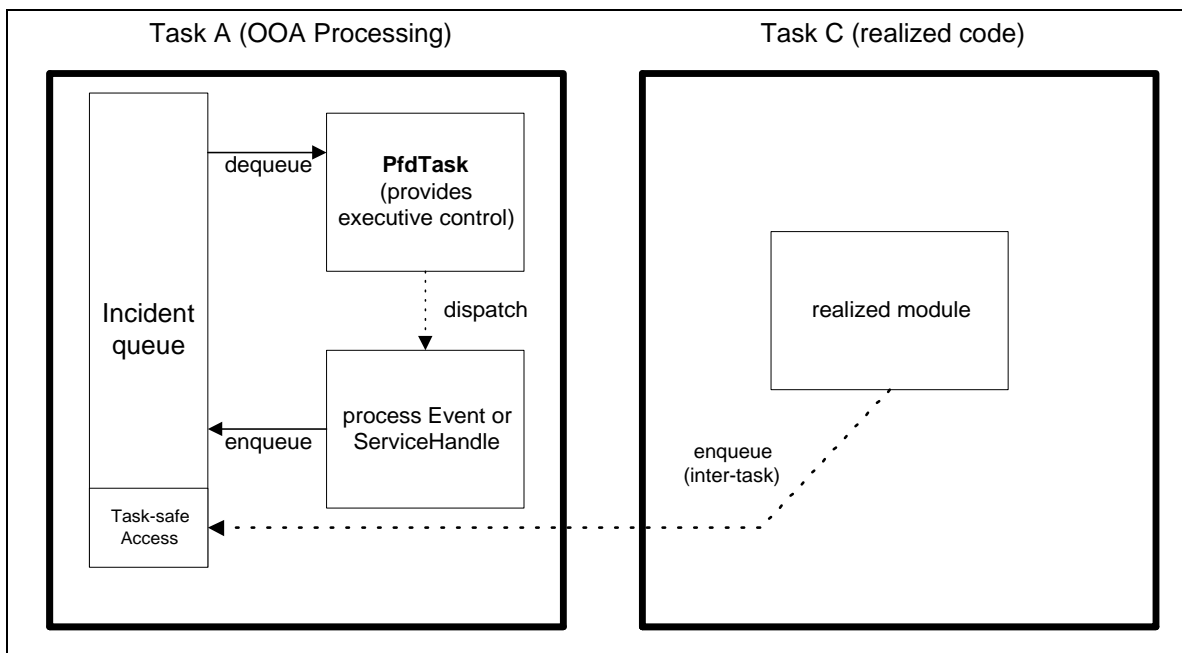
**Figure 1: Data and Control Flow within OOA Processing Task**

By providing an intertask contention mechanism on the Incident queue (called a critical section, with implementations for all supported platforms), two or more tasks running OOA Processing can interact through the protected exchange of Incidents.



**Figure 2: Two OOA Processing Tasks**

In addition, realized code in other tasks can interact with a OOA Processing task by using provided PfdTask/SW\_Task methods to add Incidents (generally ServiceHandles to domain services) to that task's Incident queue.



**Figure 3: A Realized Task Communicating with an OOA Processing Task**

Today all the base mechanisms exist to support the basic OOA Processing and protected intertask sharing of Incidents. What is missing is the ability to readily allocate subsets of an MDA system to individual separate tasks. The features described below introduce increasing extensions to these base capabilities.

## 4. Capabilities

This document outlines three levels of support of multi-task capabilities. At the base level, one or more modeled domains are allocated to a specific task. The next level allocates individual elements within the domain (domain services and classes) to tasks that may not be the base domain's task. Finally, mechanisms can be used to dynamically execute domain services and classes in new RTOS tasks - created on the fly.

### Task Labels

Tasks are identified by the following system:

- **SYS\_TASK\_** - this prefix is used to identify all tasks labels.
- **SYS\_TASK\_ID\_MAIN** - This is the default task, and anything not explicitly allocated to another task will run in this task. In multi-processor systems, all processors have a SYS\_TASK\_ID\_MAIN.
- **SYS\_TASK\_ANY** - This identifies that this model element is not allocated to a single task, and can run locally in any task that calls/invokes it.

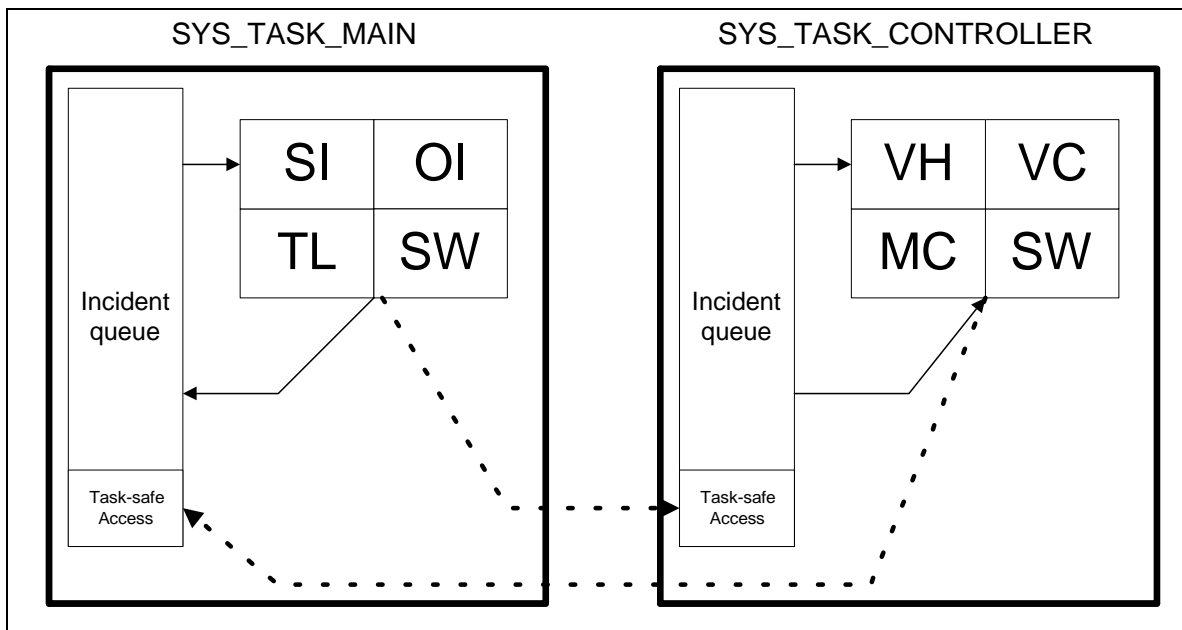
### Domain Allocation

One or more modeled domains can be allocated to a specific RTOS task. This allocation is fixed at translation (code generation) time, and the tasks will be configured at startup. It is assumed that the SYS\_TASK\_ID\_MAIN task will be one of the tasks used - it is the default task for any domain. System level initialization is run in this task.

As an example, let's assume we will deploy the CarShuffle system onto two tasks - the SYS\_TASK\_ID\_MAIN task for user interface activities, and a higher priority SYS\_TASK\_ID\_CONTROLLER task for vehicle/motor controlling functions. Our allocation of domains to tasks is:

<b>domain</b>	<b>prefix</b>	<b>TaskID</b>
MotorControl	MC	SYS_TASK_ID_CONTROLLER
OperatorInterface	OI	SYS_TASK_ID_MAIN
ScreenInterface	SI	SYS_TASK_ID_MAIN
SoftwareMechanisms	SW	SYS_TASK_ANY
TextLabels	TL	SYS_TASK_ID_MAIN
VehicleControl	VC	SYS_TASK_ID_CONTROLLER
VehicleHousing	VH	SYS_TASK_ID_CONTROLLER

**Figure 4: Domain Allocation Table for CarShuffle**



**Figure 5: CarShuffle Task Composition**

### Analysis Property Extensions

A domain property *TaskID* controls this allocation. The values are:

- SYS\_TASK\_ID\_MAIN, indicating this domain runs in the primary OOA Processing task (default)
- SYS\_TASK\_ANY, indicating this domain runs whatever task it is called from (note: any Event GENERATION or IncidentHandle (ServiceHandle) CALLs to this domain made from a realized task will be dispatched to the SYS\_TASK\_ID\_MAIN task)
- <any valid project-specific predefined task id>

For any domain with a *TaskID* that is not SYS\_TASK\_ID\_MAIN or SYS\_TASK\_ANY, the domain property *TaskPriority* may be set. The values for task priority are:

- SYS\_TASK\_PRIORITY\_HIGHEST
- SYS\_TASK\_PRIORITY\_HIGHER
- SYS\_TASK\_PRIORITY\_NORMAL (default)
- SYS\_TASK\_PRIORITY\_LOWER
- SYS\_TASK\_PRIORITY\_LOWEST

If more than one domain is allocated to the same non-MAIN/non-ANY task, their specified priorities must be the same.

The actual effect these priorities have is specific to the task scheduling support available in your target execution environment.

### **Translation Features**

The generated code automatically provides all elements needed to manage all aspects of task startup and use.

**Initialization:** The task topology is determined by a survey of the TaskID for each domain. A startup function is generated to start each task. The startup processing for each task is determined by the domain init actions for the domains allocated to each task. The system level init action is started in SYS\_TASK\_ID\_MAIN.

**ServiceHandle:** The current task identifier is specified during the creation of service handles. When a service handle is CALLED, it is routed to the Incident queue for the proper task.

**Domain Service Invocation:** When a domain service invocation is translated from PAL, the task id for the target domain will be compared to the calling domain's task id. If they are the same, a synchronous method/function invocation will be generated. If they are different, code will be generated to automatically create a service handle and route it to the Incident queue for the proper task. *Please note* – inter-task invocation of services with output parameters or a return value is not supported.

### **Mechanisms Features**

**Service Handle:** The PfdServiceHandle (SW\_Incident in C) class carries a task id.

**Startup:** The generated System::Run() method (or System\_Run() function in C) is extended to automatically invoke the appropriate generated task initialization code.

### **Analysis Implications**

This feature uses the domain boundaries already established and enforced by MDA, so there is little practical impact on the analyst. The most significant constraint is all domain services to be invoked across task boundaries must be of asynchronous form – have no output parameters or return value.



## Domain Service Allocation

For designers requiring a finer level of control, support is provided to allocate distinct domain services to specified tasks. This allocation is fixed at translation (code generation) time, and the tasks are configured and started at system startup.

An active class instance is allocated to the task it was created in. If it was statically initialized, it is allocated to the domain's default task. This only affects the Incident queue that Events destined for this instance are queued in.

As an example let's revisit the CarShuffle system, and respond to a new requirement: for safety purposes all activities that potentially support evacuation in case of building fire must be run in the SYS\_TASK\_ID\_EVACUATION task. This task is established at startup time at a higher priority (this is done by project-specific realized code and not through any model properties or analysis calls), and remains available for evacuation activities. The VehicleControl Backout service has been identified as a SAFETY capability, and must be allocated to this new task. Our allocation of domains to tasks would be the same as the previous example, except MC is now allocated to ANY to allow it to be used in both CONTROLLER and EVACUATION.

<b>domain</b>	<b>prefix</b>	<b>TaskID</b>
MotorControl	MC	SYS_TASK_ANY
OperatorInterface	OI	SYS_TASK_ID_MAIN
ScreenInterface	SI	SYS_TASK_ID_MAIN
SoftwareMechanisms	SW	SYS_TASK_ANY
TextLabels	TL	SYS_TASK_ID_MAIN
VehicleControl	VC	SYS_TASK_ID_CONTROLLER
VehicleHousing	VH	SYS_TASK_ID_CONTROLLER

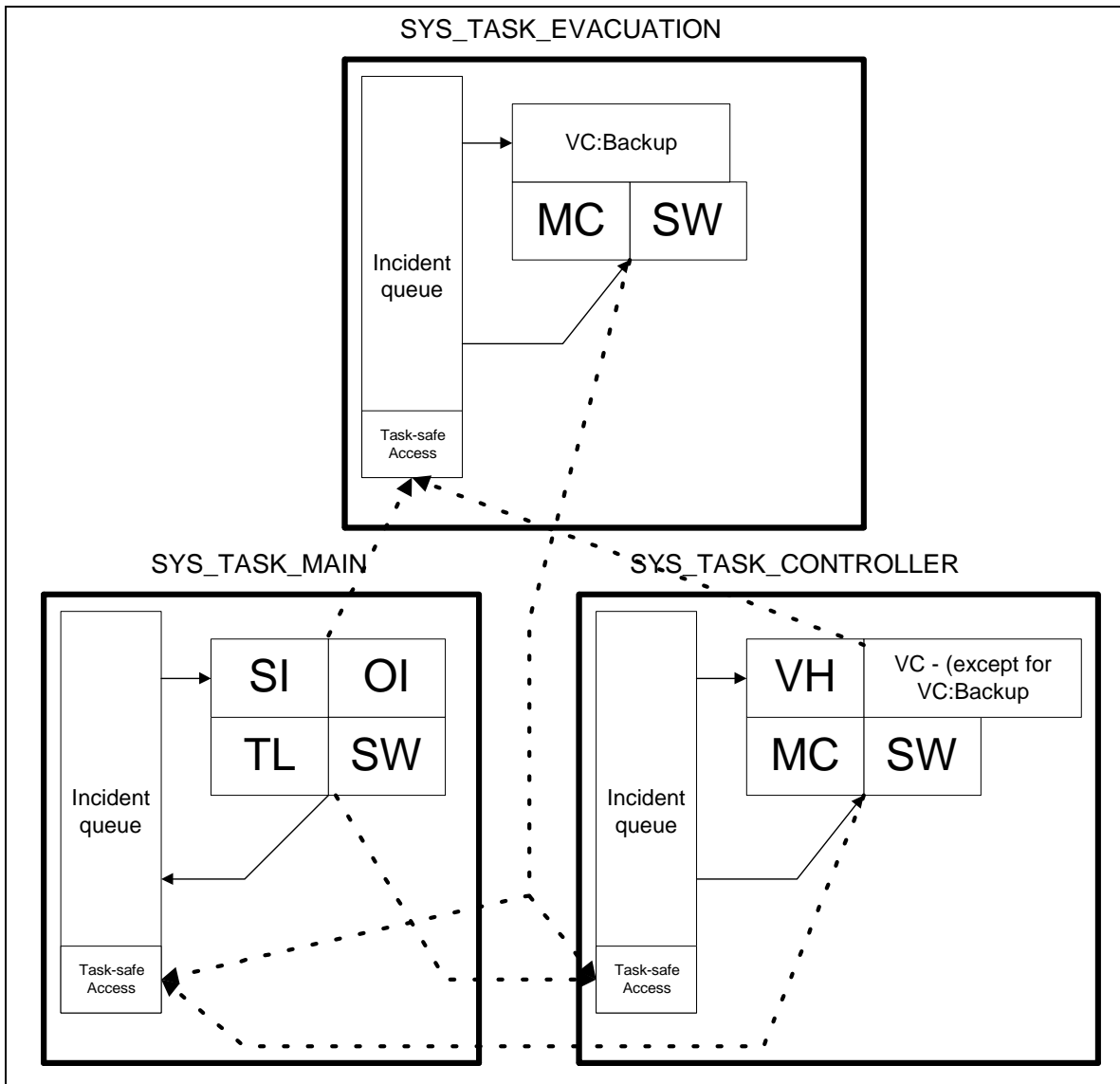
**Figure 6: Domain Allocation Table for CarShuffle with EVACUATION Task**

The base domain allocation is supplemented by this domain service allocation:

<b>service</b>	<b>TaskID</b>
VC:Backout	SYS_TASK_ID_EVACUATION

**Figure 7: Domain Service Allocation Table for CarShuffle with EVACUATION Task**

These allocations would result in this task composition:



**Figure 8: CarShuffle Task Composition with EVACUATION Task**

(All extensions below assume the completion of the Domain Allocation feature as a base.)

### **Analysis Property Extensions**

The domain service property *TaskID* defines which task this service is allocated to. The values are:

- SYS\_TASK\_ID\_MAIN, indicating the primary OOA Processing task
- SYS\_TASK\_ANY, indicating this domain runs whatever task it is called from (note: any Event GENERATION or IncidentHandle (ServiceHandle) CALLs to this domain made from a realized task will be dispatched to the SYS\_TASK\_ID\_MAIN task)
- <any valid predefined task id>

The domain property *TaskID* specified the default TaskID for a domain service.

### **Translation Features**

**Service Handle:** If a ServiceHandle is created for a domain service that has a *TaskID* specified, this id will be used. Otherwise the current task identifier will be specified.

**Domain Service Invocation:** When a domain service invocation is translated from PAL, the task id for the target domain is compared to the calling domain's task id. If they are the same, a synchronous method/function invocation is generated. If they are different, code is generated to create a service handle and route it to the Incident queue for the proper task.

**Event Generation:** When an event is generated, the task id for the event is filled in from the task id of the destination instance.

**Class and Association Accessor Contention Resolution:** Class and association accesses are examined for potential intertask conflict, and if this potential is detected, a transformation-time report is generated to identify these cases.

### **Mechanisms Features**

**Incident:** The PfdIncident class is extended to carry a task id.

**ActiveClass:** The PfdActiveClass is extended to carry a task id.

### **Analysis Implications**

While this incremental capability may not seem like a big change relative to the allocation of a complete domain to a task, it opens the door for intertask access issues within a domain. Since any generated mechanisms for intertask access contention may incur substantial accumulated run-time overhead, analysts should reduce intertask contentions and try to use domain services and events for intertask activities.

## Class Allocation

Following the release of Domain Allocation capabilities (either as an alternative to, or in conjunction with Domain Service Allocation), support is provided to allocate distinct active classes to specified RTOS tasks. This allocation is fixed at translation (code generation) time, and the tasks are at startup.

If a task is specified for an active class via *TaskID*, each instance of the class is allocated to the specified task. If no TaskID is specified for an active class, it is allocated to the domain's default task. This only affects the Incident queue that Events destined for this instance are queued in – all class services are run in the task they are invoked from.

As an example let's revisit the CarShuffle system safety requirement and now we allocate the VehicleHousing class to the SYS\_TASK\_ID\_EVACUATION task. Our allocation of domains to tasks would be the same as the previous example, except both VC and MC are now allocated to SYS\_TASK\_ANY to allow them to be used in both CONTROLLER and EVACUATION.

<b>domain</b>	<b>prefix</b>	<b>TaskID</b>
MotorControl	MC	SYS_TASK_ANY
OperatorInterface	OI	SYS_TASK_ID_MAIN
ScreenInterface	SI	SYS_TASK_ID_MAIN
SoftwareMechanisms	SW	SYS_TASK_ANY
TextLabels	TL	SYS_TASK_ID_MAIN
VehicleControl	VC	SYS_TASK_ANY
VehicleHousing	VH	SYS_TASK_ID_CONTROLLER

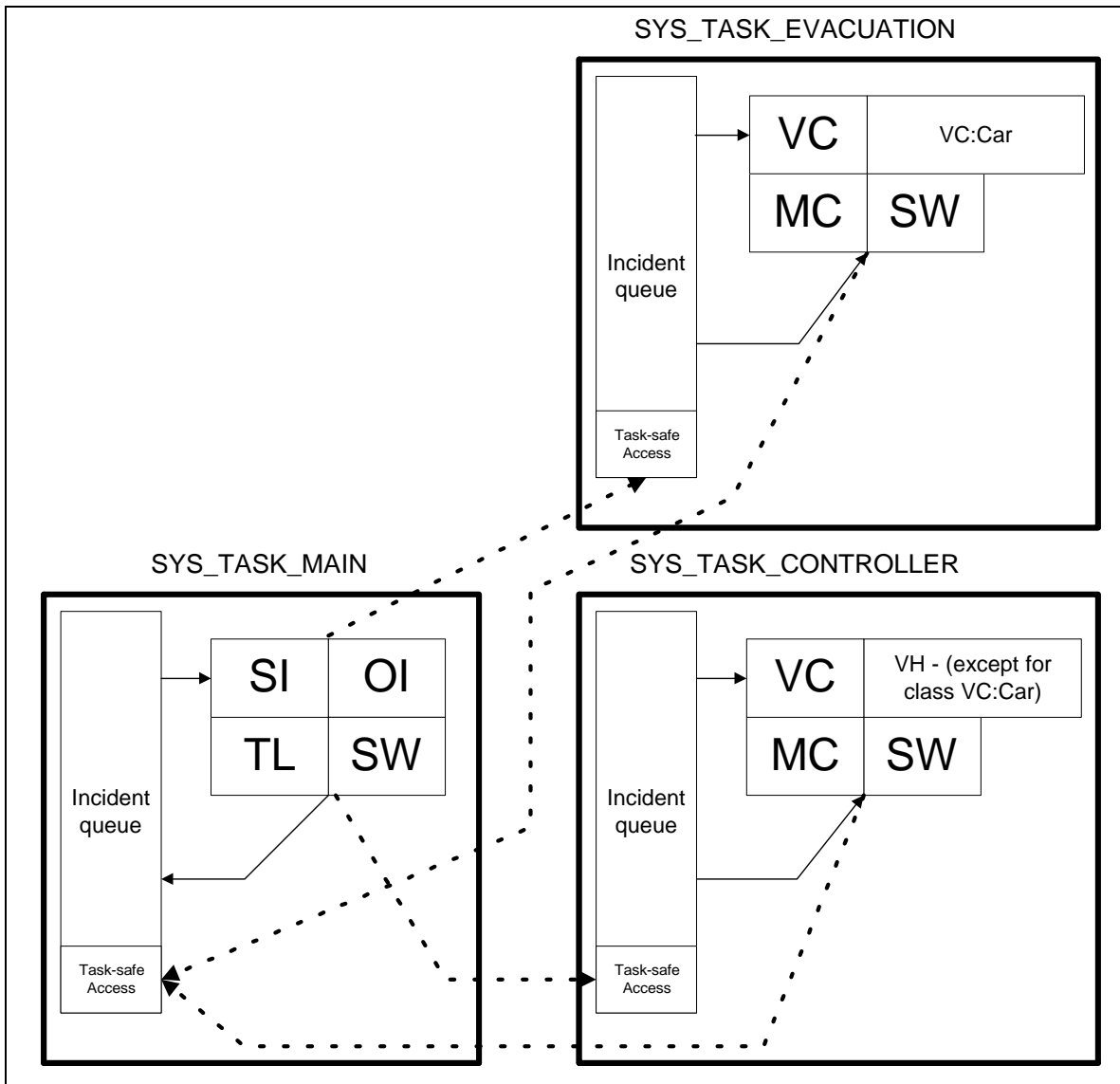
**Figure 9: Domain Allocation Table for CarShuffle with EVACUATION Task (alternative)**

The base domain allocation is supplemented by this class allocation:

<b>service</b>	<b>TaskID</b>
Vh:Car	SYS_TASK_ID_EVACUATION

**Figure 10: Class Allocation Table for CarShuffle with EVACUATION Task**

These allocations result in this task composition:



**Figure 11: CarShuffle Task Composition with EVACUATION Task (class-based)**

### Analysis Property Features

The class property *TaskID* defines which task this service is allocated to. The values are:

- **SYS\_TASK\_ID\_MAIN**, indicating the primary OOA Processing task
- **SYS\_TASK\_ANY**, indicating this class runs whatever task it is called from (note: any Event GENERATION or IncidentHandle (ServiceHandle) CALLs to this domain made from a realized task will be dispatched to the **SYS\_TASK\_ID\_MAIN** task)
- <any valid predefined task id>

The domain property *TaskID* will provide the default value if no value is specified for the domain service property.

### ***Translation Extensions Features***

**Service Handle:** If a ServiceHandle is created for a class service for a class that has a *TaskID* specified, this id is used. Otherwise the current task identifier is specified.

**Event Generation:** When an event is generated, the task id for the event is filled in from the task id of the destination instance.

**Class and Association Accessor Contention Resolution:** Class and association accesses are examined for potential intertask conflict, and if this potential is detected, a transformation-time report is generated to identify these cases.

### ***Mechanisms Extensions***

**Incident:** The PfdIncident class is extended to carry a task id.

**ActiveClass:** The PfdActiveClass is extended to carry a task id.

### ***Analysis Implications***

While this incremental capability may not seem like a big change relative to the allocation of a complete domain to a task, it opens the door for intertask access issues within a domain. Since any generated mechanisms for intertask access contention may incur substantial accumulated run-time overhead, analysts should reduce intertask contentions and try to use domain services and events for intertask activities.

## **Dynamic Domain Service Allocation**

*(Please review the Dynamic tasking technical note for details on how to create/start and use RTOS tasks on the fly.)*