EventStudio System Designer 6 is a systems engineering tool that uses sequence diagrams to model complex system interactions.
EventStudio System Designer
6
USER'S MANUAL

Contents

A NEW APPROACH TO SYSTEMS ENGINEERING ............................................................... 1
   Powerful but Simple Modeling Language ............................................................... 1
   Focus on Systems Engineering not Diagram Layout ................................................. 2
   Catch Design Errors Early in the Lifecycle ............................................................. 2
   Zoom In and Out of Different Levels of Abstraction ............................................... 2
   Model Multiple Scenarios ....................................................................................... 3
   Wide Variety of Applications ................................................................................... 3
   Slice, Dice and Analyze Your Model ...................................................................... 4
   Customize Your Diagrams ...................................................................................... 5

GETTING STARTED ......................................................................................................... 7
   Your First Scenario Project ..................................................................................... 7
   Modify the FDL File and Regenerate Document .................................................... 10
   Learn More About FDL ......................................................................................... 12
   Explore More ........................................................................................................ 13

FDL TUTORIAL ................................................................................................................ 14
   Introduction to the Language .................................................................................. 14
   Comments, Remarks and Issues ............................................................................. 15
   Object Management and Message Parameters ..................................................... 16
   Timer and Resource Management ........................................................................ 18
   Environment Interactions, States and Actions ....................................................... 19
   Defining Types for Modules, Components and Objects ......................................... 20
   Case, If-Else-EndIf and Page Break Statements ..................................................... 22
   Explore More ......................................................................................................... 25

VISUAL GUIDE TO FDL ................................................................................................ 26
   Architecture Definition ......................................................................................... 26
   Messages ............................................................................................................... 27
   Message Tagging ................................................................................................. 28
   Object Interactions ............................................................................................... 29
   Object Creation and Deletion ................................................................................. 30
   Timers ................................................................................................................... 31
Actions ................................................................. 32
Single Action......................................................... 32
Multiple Action ...................................................... 33
Resource Allocations ................................................ 34
State Transitions..................................................... 35
Sequence Groupings ................................................. 36
Remarks................................................................... 37

FDL REFERENCE .................................................................. 38
Architecture and Feature Definition.................................. 38
System declaration ........................................................ 38
Subsystem declaration .................................................. 39
Module declaration ...................................................... 41
Component declaration ................................................ 42
Eternal Object declaration ............................................. 44
Dynamic Object declaration .......................................... 46
Type declaration .......................................................... 49
Environment declaration .............................................. 50
Feature statement ....................................................... 50
Multiple Scenario Modeling ........................................... 51
Case statement ............................................................ 51
Leg statement ................................................................ 52
If-Else-Endif statement ............................................... 53
If-Endif statement ........................................................ 54
Goto and Label statements .......................................... 55
Messages .................................................................... 56
Message statement ...................................................... 56
Chain statement .......................................................... 58
Multicast statement ..................................................... 60
Object Interactions ...................................................... 61
Invokes and Returns statements .................................... 61
Object Creation and Deletion ......................................... 62
Create statement ......................................................... 62
Delete statement ......................................................... 63
Timers ........................................................................ 63
Timer Start (One Shot and Periodic) statement ................. 63
Timer Restart statement ............................................... 64
Timer Stop statement .................................................. 65
Timeout statement ........................................................ 65
Time Passage statement .............................................. 65
Actions ....................................................................... 67
Action statement ........................................................ 67
Action Begin statement ......................................................... 68
Action End statement .......................................................... 69
Resource Allocations ............................................................ 70
Resource Allocate statement .................................................. 70
Resource Free statement ....................................................... 71
State Transitions ..................................................................... 72
State Change statement ......................................................... 72
Sequence Groupings ................................................................ 73
Sequence Block Statement ...................................................... 73
Remarks .................................................................................. 75
Remark statement ..................................................................... 75
Block Remark statement ........................................................ 75
Comment statement ............................................................... 76
Heading statement .................................................................... 76
Issue statement ........................................................................ 76
Preprocessor ............................................................................. 77
Styles, Themes and Document Layout ...................................... 78
Style declaration ....................................................................... 78
Default Styles .......................................................................... 81
RegularExpression Style Filters .................................................. 83
Theme declaration ...................................................................... 83
Page Break statement .............................................................. 85
Horizontal and Vertical Separator statements .......................... 86

HOW TO ................................................................................. 87
How to Add a New Scenario ....................................................... 87
How to Add a Document .......................................................... 88
How to Add Review Comments to a Sequence Diagram ............ 93
How to Export to Microsoft Word .............................................. 94
How to Rename a Scenario ....................................................... 94
How to Reorder Scenarios ......................................................... 95
How to Delete a Scenario ........................................................ 95
How to Create a Scenario Project from Existing FDL Files ....... 95
How to Create a New FDL File .................................................. 96
How to Generate Documents from Command Line .................. 97
How to Rename a Document ..................................................... 97
How to Delete a Document ....................................................... 98

WHAT’S NEW IN EVENTSTUDIO SYSTEM DESIGNER 6 ....................... 99
Message Endpoint Tagging ....................................................... 99
Specifying the Message Tags ................................................... 99
Defining a Tag Style ............................................................... 99
Tag Export in XML ................................................................. 100
Object Interaction Enhancements .............................................................................................................. 112
  Hierarchically Classify Object Interactions ............................................................................................. 112
  Generate Object Interaction Diagrams at Different Levels of Abstraction ............................................... 112
  Specify Object Creation Parameters ........................................................................................................ 113
Large Scale System Modeling ....................................................................................................................... 113
  Share a Single Instance Axis between Multiple Objects ............................................................................. 113
  Anonymous Object Create and Delete ....................................................................................................... 113
Model Continuous Actions ........................................................................................................................... 113
Command Line Enhancements ..................................................................................................................... 113
  Specify Conditional Defines at Command Line ......................................................................................... 113
  Define Include Paths from Command Line ............................................................................................... 113
Other Enhancements .................................................................................................................................. 114
  Object wise Summary Enhancements ....................................................................................................... 114
  Unit Test Procedure Enhancements ......................................................................................................... 114
  Hyperlink Support Extended to all Action Statements ............................................................................ 114
  Simplified EMF File Update ...................................................................................................................... 114
  Component Box in Light Weight Header ................................................................................................... 114
  Model Passage of Time ............................................................................................................................... 114
  Control Spacing with Separators ............................................................................................................ 114
REGULAR EXPRESSIONS .......................................................................................................................... 116
  Regular Expression Examples .................................................................................................................... 116
  Regular Expression Rules .......................................................................................................................... 117
COLORS ......................................................................................................................................................... 119
A NEW APPROACH TO SYSTEMS ENGINEERING

EventStudio System Designer 6 is a systems engineering tool that uses sequence diagrams to model complex system interactions.

- **Powerful but Simple Modeling Language**
- **Focus on Systems Engineering not Diagram Layout**
- **Catch Design Errors Early in the Lifecycle**
- **Zoom In and Out of Different Levels of Abstraction**
- **Model Multiple Scenarios**
- **Wide Variety of Applications**
- **Slice, Dice and Analyze Your Model**
- **Customize Your Diagrams**

**Powerful but Simple Modeling Language**

EventStudio uses the Feature Description Language (FDL) as the modeling language. With FDL you can:

- Specify the system architecture in terms of system, subsystem, module, component and objects.
- Represent message interactions with detailed parameter specification.
- Specify multicast and broadcast operations.
- Model timer start, stop, restart and expiry.
- Model resource allocation and freeing.
- Dynamically create and delete objects.
- Define multiple scenarios without repeating the common parts.
- Represent state transitions.
- Model single step and multi-step object actions (e.g. dial-tone start and stop).
- Specify inline and block remarks to document the design.
- Get full control over the rendered color and font with styles and themes.
- Share common parts of the design by defining include files and macros.
- Maintain multiple variations of a design using conditional “compilation”.

```plaintext
module: UE, RAN, CN
component: NAS in UE, AS in UE
component: eNodeB in RAN, MME in CN
eternal: Radio in eNodeB, MAC in eNodeB
feature "LTE call flow"
  RACH : AS -> Radio
  (* RACH received from UT *)
  "RACH Indication" (frame, timing, doppler) : Radio -> MAC
  (* Pass message to MAC. *)
  MAC takes action "Handle msg"
  (* Invoke handler. *)
endfeature
```
Focus on Systems Engineering not Diagram Layout

Why waste developer time in diagram layout when EventStudio can automatically layout the diagrams?

- Do not waste precious development time on clumsy graphical editors that do not scale with increasing complexity
  - As the design gets complicated, graphical editors get increasingly difficult it with the large number of nodes involved.
- A text based development language is closer to the mental model developers use when designing a system.

Catch Design Errors Early in the Lifecycle

EventStudio analyzes the design and catches design errors in success and failure scenarios. Examples of some design errors caught by EventStudio:

- Resource leaks and buggy resource management
- Missed object creation and deletion
- Missed action completion for an action
- Free running timer and invalid timer management

Zoom In and Out of Different Levels of Abstraction

FIGURE 1 DETAILED THREE TIER DIAGRAM

FIGURE 2 ZOOM OUT TO TWO TIER HIGH LEVEL DIAGRAM (COMPONENT LEVEL INTERACTIONS)
Hierarchically decompose the design into multiple levels
Generate diagrams at different levels of abstraction from the same model.
Get customers, marketing team, system architects, development managers, developers and the test team on the same page. For example:
- The marketing team refer to the diagram that describes only inter-system level interactions
- Developers focus of detailed object interactions in a subsystem
- System architects use diagrams that involve system and subsystem interactions.

Model Multiple Scenarios
System architects often generate use cases and sequence diagrams only for basic scenarios. A large number of scenarios are simply left out as it is not practical to document and manage them.

EventStudio addresses these problems with built in support for multiple scenario modeling.

- When designing multiple scenarios, just provide details on how the new scenario differs from the base scenarios.
- Common parts shared in multiple scenarios need to be defined only one.
- Automatically generate documents for a large number of success and failure scenarios

Wide Variety of Applications
Call Flow Diagrams and Message Sequence Charts

- Model call signaling interactions and call state transitions
- Specify call timer start, stop and expiry
- Define multiple successful and failure call scenarios

Object Sequence Diagrams

- Model object interactions and analyze them at different levels
- Specify multiple legs of execution

Use Case Diagrams

- Model use case interactions with sequence diagrams
- Model a large number of success and failure scenarios
- Generate a high level and detailed use case description
Slice, Dice and Analyze Your Model

- Generate XML output and transform it into custom formats with XSLT. With custom XSLT you can generate:
  - State transition diagrams
  - Class diagrams
  - Collaboration diagrams
  - Test code

- Analyze your model with diagrams that:
  - Extract interfaces for a specified entity
  - Include only the messages that match a specified regular expression
  - Summarize the interactions object wise
  - Specify the unit tests for a particular entity
Customize Your Diagrams

EventStudio allows you to customize the diagrams in myriads of ways:

**FIGURE 4 MULTI TIER HEADER**

**FIGURE 5 SINGLE TIER HEADER**

**FIGURE 6 HEADER WITH IMAGES**
FIGURE 8 SUMMARY SEQUENCE DIAGRAM

FIGURE 7 DETAILED SEQUENCE DIAGRAM
GETTING STARTED

Your First Scenario Project

Follow the steps below to get started with your first scenario project.

1. Download and install EventStudio System Designer (45 day free trial). (Download link: http://www.eventhelix.com/EventStudio/Download.htm)

2. Click on the New and Recent Document icon.

3. Select Create New Project from Template and click OK.
4 Select Starter Scenario Project (Simple) and click Next.

5 Change the Project and FDL names and click Finish.
After you click Finish, EventStudio generates a PDF sequence diagram from the initial contents.
Modify the FDL File and Regenerate Document

You have just created a scenario project from a standard template. We will now edit the generated FDL and regenerate the sequence diagram.

7. Double click on the FDL entry to open the corresponding FDL file.

8. Modify the FDL file contents in the editor window. For this tutorial, add the lines shown in the shaded box.

9. FDL syntax is intuitive. You may also refer to the user’s manual.

10. Click on the Quick Save and Display icon. EventStudio automatically updates the PDF sequence diagram.
The newly added message statement has been added to the PDF sequence diagram.
Learn More About FDL

Now let's learn more about the Feature Description Language (FDL) and the FDL development in EventStudio.

#include "inc.fdl"

/* TODO: Replace Module_A architecture with your declarations. */
module : Module_A
component : Proc_A_1 in Module_A, Proc_A_2 in Module_A
eternal : Object_A_1 in Proc_A_1, You in Proc_A_1
eternal : Object_A_2 in Proc_A_2

/* TODO: Replace Module_B architecture with your declarations. */
module : Module_B
component : Proc_B_1 in Module_B, Proc_B_2 in Module_B
eternal : Object_B_1 in Proc_B_1, Me in Proc_B_1
eternal : Object_B_2 in Proc_B_2

/* TODO: Replace the body with your implementation. */

{MyTheme} feature "Hello Goodbye"

"My Message" (param="A") : Me -> You
(* Added a new message. *)

/* TODO: Add your message interactions */

Goodbye(reason="Nothing in Particular") : You -> Me
(* You say Goodbye... *)

Hello : Me -> You (* ...and I say Hello! *)

You, Me take action "Hello World!"

MSG_AND_ACK(Msg, Me, Object_B_1)
METHOD_AND_RETURN(StartOperation, Me, Object_B_1)
endfeature
Explore More

We have just scratched the surface of what you can do with EventStudio. Please refer to the following tutorials to learn more:

- FDL Tutorial
- Visual Guide to FDL

Click the Generate All icon to generate all the documents.

Double click on the documents here. Some of diagrams in this project are:
- Component and module level sequence diagrams
- Collaboration diagram
- EMF sequence diagram for export to MS Word.

More examples.
FDL TUTORIAL

The software development involves the design state followed by coding. The design stage describes the system using informal methods like text documents, figures. The output of this stage cannot be verified for correctness by tools.

The coding stage happens to be the first formal description of the system. The output of this stage can be verified by using compilers and linkers.

This represents a sort of impedance mismatch between the two development processes. Design is largely informal while coding is completely formal. The Feature Description Language (FDL) tries to bridge this gap by introducing a semi-formal development system for design. It tries to incorporate features from both the stages:

- FDL documents allow the user to express the system even when all the details of the system have not been defined.
- FDL documents allow the user to review the correctness of the system design by running an automated review process.

Introduction to the Language

A very simple FDL program is shown below. It shows modules and components defined in the system. Message interactions between components are shown enclosed in the feature-endfeature block. We will be building on this example as we go along with basic and advanced tutorials. The FDL file and documents generated at each stage are available in the Tutorial directory of the EventStudio installation.

```
1. module : customer, exchange
2.     component : phone in customer
3.     component : frontend in exchange, core in exchange
4.     feature "Call Setup"
    offhook : phone -> frontend
    dialtone : frontend -> phone
    digits : phone -> frontend
    setup_call : frontend -> core
    setup_complete : core -> frontend
    ringback : core -> phone
   endfeature
```

This program defines the message exchanges between a customer and a telephone exchange. The customer and the exchange have been declared with the module declaration.

1. The component statements in the next two lines define different entities within the customer and the exchange. Here the customer contains a phone and the exchange contains a frontend and a core component. This relationship is specified using the in keyword.

2. The feature-endfeature block follows the declarations in FDL. A title for the feature is included in the feature declaration. The feature...endfeature block encloses all the feature interactions between the customer and the exchange.
3. Message interactions have been enclosed within the feature-endfeature block. The first `message` interaction in the sequence sends an offhook message from the phone to the frontend component. This is followed by other message interactions involved in call setup. Messages are represented as arrows from the source to the destination.

**Note:** FDL also supports a multicast statement for modeling message interactions where a single message is simultaneously received by multiple entities. This statement has not been covered in this tutorial.

Examine the Tutorial.pdf file at this point to see the correspondence between the FDL file and PDF output.

---

**What are Modules, Components and Objects?**

FDL allows you to partition your system into a multi level hierarchy. When using a three level hierarchy, the highest level is modules. Modules contain components and components contain eternal and dynamic objects. The selection of modules, components and object is best explained with examples.

**Acme Inc. Recruiting**

Modules are Recruiters, Acme_Inc, Media, Other_Company etc.

Components contained in Acme_Inc are the various departments in the company, e.g. Finance, HR, IT.

Objects contained in the HR department are HR_Secretary, Recruitment_Specialist.

**Highway System**

Modules are Highways, EntryRamp, TollBooth etc.

Components contained in Highway are Cars, Trucks, Motorbikes etc.

Object contained in a Car are steering, brakes, engine etc.

---

**Comments, Remarks and Issues**

This part of the tutorial introduces you to the various types of comments and remarks that are supported by FDL. The following comment types are supported:

- C-Type comments
- Remarks
- Block Remarks
- Issue statement

Standard C-type comments are supported for programmer documentation. Remarks and Block Remark comments allow the user to explain the feature flow. There output will appear in the PDF documents. Issue statements are used to keep track of unfinished sections and review comments. Issue statements output appears in the PDF output as well as Output Window at the bottom of the screen.
module: customer, exchange
component: phone in customer
component: frontend in exchange, core in exchange

1. /* Originating Call Setup */
   feature "Call Setup"
   offhook : phone -> frontend
2. (* Subscriber lifts the phone to initiate a call *)
   dialtone : frontend -> phone
digits : phone -> frontend
setup_call : frontend -> core

3. issue "Call setup parameters have not been defined"
setup_complete : core -> frontend

4. [* Setup completed, waiting for called subscriber to be rung *]
   ringback : core -> phone
endfeature

1. FDL supports C-type comments enclosed in /* and */ i.e. /* comment */. Multi-line comments can be placed between any FDL statements. EventStudio will ignore these comments during the initial review. C-type comments have no impact of the FDL output documents.

2. FDL supports remarks enclosed in (* and *) i.e. (* remark *). The remarks are always printed on the right side remark column of PDF documents. FDL associates remarks with the previous FDL statement. In this case, the remark corresponds to the message statement sending an offhook from the phone to the frontend. Remarks should generally be defined for almost every statement. This helps the reader in quickly understanding the feature.

3. An issue statement can be added whenever the author has some unresolved issues. These statements are printed in bold on in the remarks column. Issue statements are reported in the output window whenever the author reviews the document. This allows the author to quickly identify if any issues are unresolved. This statement may also be used to provide review comments on an FDL file.

4. FDL supports block remarks enclosed in [* and *] i.e. [* block remark *]. Block remarks are always printed along the complete width of the PDF output page. The block remarks do not associate with any FDL statement. Use block remarks to show milestones in the execution of a feature. They may also be used to define large remarks that would not fit as a normal remark.

Notes:

- Block remarks as [* block remark *] and remarks as (* remark *) can be multi-line.
- Block remarks also support division of text into multiple paragraphs.

Object Management and Message Parameters

We have seen earlier that FDL allows the developer to describe the system by representing it in terms of modules and components. FDL also supports definition of objects inside the components. Objects could be eternal or dynamic. Eternal objects are created at system startup and they exist throughout the life of the system. Dynamic object on the other hand are created and destroyed dynamically. The system represents eternal objects with instance axis that runs from the beginning of the document to the end. The instance axis for dynamic objects runs from the creation time to the deletion time.
In Call3.fdl we have added call_mgr and call objects to the frontend component. Once objects are defined on a component, all the messages should specify the objects on the component. Thus references to the frontend component have been replaced with references to call_mgr and call objects. In addition to this, definition of objects allows us to specify the detailed message handling on the frontend component.

Additionally, the tutorial also demonstrates how message parameters can be specified in FDL. Message parameters can be added to any message interaction.

```plaintext
module : customer, exchange
component : phone in customer
component : frontend in exchange, core in exchange
1. eternal : call_mgr in frontend
2. dynamic : call in frontend
/* Originating Call Setup */
feature "Call Setup"
3. offhook : phone -> call_mgr
   (* Subscriber lifts the phone to initiate a call *)
4. call_mgr creates call
   offhook : call_mgr -> call
   dialtone : call -> phone
   digits : phone -> call
5. setup_call(digits="1-800-433-444", mode=NORMAL) : call -> core
   setup_complete : core -> call
   [* Setup completed, waiting for called subscriber to be rung *]
   ringback : core -> phone
   onhook : phone -> call
   call_over : call -> call_mgr
6. call_mgr deletes call
endfeature
```

1. This statement declares an **eternal** object call_mgr running on the frontend component. Use this statement to declare objects that are created at system startup and never die during the system operation (hence the name eternal).

2. A **dynamic** object is declared in this statement. Use this statement to declare objects that will be created and deleted dynamically by the system.

3. We have now defined two objects in the frontend component. Thus all message interactions need to specify the object that will be receiving the message. In this case, the message destination has been changed from frontend component to call mgr eternal object.

4. A dynamic object needs to be created explicitly. Here a call object has been created. Use the `create` statement to create dynamic objects.

5. This statement just adds more detail to the setup_call message. The **message parameters** digits and mode have been specified. Message parameters are printed below the message arrow. The message parameter values are specified as a string in one case. Note that the message name, field name and the field value can all be defined as strings or identifiers. (Also note that specifying the parameter value is optional).
6. This statement shows the call_mgr deleting the call. Typically a dynamic object created in a feature should be deleted before the feature ends (EventStudio will warn you if you don’t do so). Use the delete statement when a dynamic object needs to be deleted. The delete statement also allows a self delete (i.e. call deletes call).

**Timer and Resource Management**

This section introduces the timer and resource management statements in FDL.

FDL supports starting, stopping and timeout for timers by eternal and dynamic objects. Using the FDL timer management statements can help you identify stray timer problems (i.e. the timer was not stopped when it was supposed to). EventStudio issues a warning if it finds that timers have been left running at feature end. For dynamic objects, a warning will be issued if timers are left running at object deletion.

Resource allocation and de-allocation can be carried out by eternal and dynamic objects. FDL resource management statements can help you catch resource leaks. EventStudio issues a warning when it finds that resources have not been de-allocated at feature end. For dynamic objects, a warning will be issued if all resources allocated by the object have not been freed at object deletion.

```plaintext
module : customer, exchange
component : phone in customer
component : frontend in exchange, core in exchange
eternal : call_mgr in frontend
dynamic : call in frontend

/* Originating Call Setup */
feature "Call Setup"
offhook : phone -> call_mgr
(* Subscriber lifts the phone to initiate a call *)
1. call_mgr allocates "Subscriber Terminal"
call_mgr creates call
offhook : call_mgr -> call
dialtone : call -> phone
2. call starts await_first_digit_timer
digits : phone -> call
3. call stops await_first_digit_timer
4. call starts more_digits_timer
5. timeout more_digits_timer
setup_call(digits="1-800-433-444", mode=NORMAL) : call -> core
setup_complete : core -> call
[* Setup completed, waiting for called subscriber to be rung *]
ringback : core -> phone
onhook : phone -> call
call_over : call -> call_mgr
call_mgr deletes call
6. call_mgr frees "Subscriber Terminal"
endfeature
```

1. The Call Manager allocates a "Subscriber Terminal" resource before the call is started on that terminal. This demonstrates the resource allocate statement in FDL.
2. Here the start timer statement is used to start a timer. The Call object has just fed dialtone to the customer and it will wait for a limited time for the subscriber to dial the first digit.
3. Some digits have been received from the subscriber, so the first digit timer is stopped. This is represented by the stop timer statement in FDL.
4. Call starts the more_digits_timer awaiting more digits from the customer.
5. The customer does not dial any more digits; this results in timeout of the more_digits_timer. This is represented by the timeout statement. (Note that all timers should be stopped or they should result in a timeout. EventStudio will issue a warning if a timer is left running when the feature ends. In addition to this EventStudio will issue a warning when timers are left running at the time of deleting a dynamic object).

6. The call has ended, so the "Subscriber Terminal" resource can now be freed up. This is accomplished by using the resource free statement in FDL.

**Note:** FDL also supports timer restart and periodic timers. Details can be found in **Timers**.

### Environment Interactions, States and Actions

In this section we will cover the following topics:

- **Message interactions with the Environment**
- **State Change statement**
- **Action statements**

FDL allows the user to depict message interactions with the external world. This can be done by sending messages to the left environment (env_l) or right environment (env_r). As the names suggest, env_l and env_r are assumed to be at the left and right edges of the document. Environment interactions can be very useful in clearly spelling out what is external to the system being designed.

State changes by eternal and dynamic objects can be represented in FDL. State changes are shown as boxes with angular bracket edges (i.e. < >). Use state change statements during the detailed design phase.

Action statements allow the designer to express specific actions taken by objects. The actions represent the internal operations carried out by the object like "updating a variable", "calling a routine". FDL can also be used to depict more complicated actions that involve a distinct action begin and end. This could be used to depict actions like "starting and stopping a motor", "feeding and removing a tone".

```plaintext
module : customer, exchange
component : phone in customer
component : frontend in exchange, core in exchange
 eternal : call_mgr in frontend
dynamic : call in frontend
/* Originating Call Setup */
feature "Call Setup"

1. pick_up_phone : env_l -> phone
   offhook : phone -> call_mgr
   (* Subscriber lifts the phone to initiate a call *)

2. call_mgr takes action "Check Customer Privileges"
   call_mgr allocates "Subscriber Terminal"
   call_mgr creates call

3. call state = "Idle"
   offhook : call_mgr -> call

4. call begins action "Feeding Dialtone"
   dialtone : call -> phone
   call state = "Awaiting Digits"
   call starts await_first_digits_timer
```
dial_digits : env_l -> phone
digits : phone -> call

5. call ends action "Feeding Dialtone"
call stops await_first_digit_timer
call starts more_digits_timer
timeout more_digits_timer
setup_call(digits="1-800-433-444", mode=NORMAL) : call -> core
call state = "Awaiting Call Setup"

6. call_setup_request : core -> env_r
call_setup_response : env_r -> core
setup_complete : core -> call
call state = "Awaiting Answer"
[* setup completed, waiting for called subscriber to be rung *]
ringback : core -> phone
hang_up : env_l -> phone
onhook : phone -> call
call_over : call -> call_mgr
call_mgr deletes call
call_mgr frees "Subscriber Terminal"
endfeature

1. This statement shows a message received from the left environment. The left environment is assumed to be at the left edge of the document.

2. Here the call_mgr takes a specific action. Use the action statement to show different actions taken by an object. The action statement allows you to add implementation details to the design document.

3. Here a call state change has been shown. Use the state change statement to specify the different states taken by the state machine implementing the object. State change statements have been added to partition the sequence of actions into individual states. State change statements have been introduced at multiple places in Call5.fdl.

4. This statement shows an action being started by the object. This signifies that action "Feeding Dialtone" has been initiated. The action begin statement specifies the start of an action, while the action statement introduced in (2) specifies an action that can be completed in a single step.

5. The "Feeding Dialtone" action end is shown here. This marks the end of the action started in (4). The action end statement should be used to end all started actions that have been started with action begin. EventStudio will warn if an action end does not complete an action that was started.

6. This statement shows a message being sent to the right environment. The right environment is assumed to be at the right edge of the document.

**Defining Types for Modules, Components and Objects**

Many times in Embedded and Distributed System Design you would have come across scenarios where two instances of an object are performing different roles in the execution of a feature. To handle such cases we need to have a way to specify that the two instances belong to the same object. FDL accomplishes this with specification of types.

Consider our running example from the Basic tutorial; we have just defined the message interactions for one end of the call. The message interactions to notify the other subscriber have not been covered. This is
remedied in the FDL given below. Another customer and frontend component have been added. Also a second instance of a call and call_mgr objects have been defined. Type definitions have been used to declare that the newly introduced entities are really instances of the modules, components and objects that we have already covered.

It may be noted that defining the types will have no visual impact on the PDF Sequence Diagrams. The main difference will be seen in definition of Interface Documents and Unit Test Procedures. For example, EventStudio can generate the following documents from the FDL file in our example:

- All interactions involving call object. The output file will include message interactions involving call1 or call2.
- All interactions involving call2 object. The output file will include message interactions involving call2. Message interactions involving call1 only will be excluded.
- All interactions involving frontend components. This output file will include message interactions that involve frontend1 or frontend2.
- All interactions involving frontend1 component. This output file will include message interactions that involve frontend1. Message interactions involving frontend2 only will not be included.

```plaintext
1. module : customer1:customer, exchange, customer2:customer
2. component : phone1:phone in customer1, phone2:phone in customer2
3. component : frontend1:frontend in exchange, core in exchange
4. component : frontend2: frontend in exchange
5. eternal : call_mgr1:call_mgr in frontend1
   eternal : call_mgr2:call_mgr in frontend2
6. dynamic : call1:call in frontend1, call2:call in frontend2
   /* Originating Call Setup */
   feature "Call Setup"
   pick_up_phone : env_l -> phone1
   offhook : phone1 -> call_mgr1
   (* Subscriber lifts the phone to initiate a call *)
   call_mgr1 takes action "Check Customer Privileges"
   call_mgr1 allocates "Subscriber Terminal"
   call_mgr1 creates call1
   call1 state = "Idle"
   offhook : call_mgr1 -> call1
   call1 begins action "Feeding Dialtone"
   dialtone : call1 -> phone1
   call1 state = "Awaiting Digits"
   call1 starts await_first_digit_timer
   dial_digits : env_l -> phone1
   digits : phone1 -> call1
   call1 ends action "Feeding Dialtone"
   call1 stops await_first_digit_timer
   call1 starts more_digits_timer
   timeout more_digits_timer
   setup_call(digits="1-800-433-444", mode=NORMAL) : call1 -> core
   call1 state = "Awaiting Call Setup"
7. call_setup_request : core -> call_mgr2
```
call_mgr2 creates call2

call_setup_request : call1_mgr2 -> call2
ring_phone : call2 -> phone2
ringing : phone2 -> env_r
call2 state = "Ringing Subscriber"
call_setup_response : call2 -> core
setup_complete : core -> call1
call1 state = "Awaiting Answer"
[* Setup completed, waiting for called subscriber to be rung *]
ringback : core -> phone1
hang_up : env_l -> phone1
onhook : phone1 -> call1
release_call : call1 -> call2
stop_ring : call2 -> phone2
released_call : call2 -> call1_mgr2
call1_mgr2 deletes call2
released_call : call1_mgr2 -> call1
call_over : call1 -> call_mgr1
call_mgr1 deletes call1
call_mgr1 frees "Subscriber Terminal"

7.

1. This statement extends the original module declaration to define types for customer1 and customer2. The statement declares that customer1 and customer2 are both of the type customer. Note that no type is specified for the exchange module, as there is only one instance of this module.

2. phone1 and phone2 components are declared in this statement. Both the components are of type phone.

3. Here is another example of component type declaration. frontend1 component is declared to be of type frontend. frontend2 declared on the next line is also of type frontend. (Note that the names frontend1 and frontend2 have been used just as a convention. You could equally well use orig_frontend and term_frontend or any other names. EventStudio will just look at the types to identify that both the components are of the same type).

4. This line and the next line declare call_mgr1 and call_mgr2 eternal objects. Both objects are of the type call_mgr.

5. Types for dynamic objects can be specified in the same way. This line declares the call1 and call2 objects. Both the objects have a type call.

6. Most of this example has been changed a little to replace names like phone, customer, and call with phone1, customer1 and call1. The names phone, customer and call etc. are used to define the types.

7. New statements have been added to the FDL to specify the handling of the other end of the call.

**Case, If-Else-Endif and Page Break Statements**

Embedded and Distributed feature design involves considering multiple success as well as failure scenarios in the execution of a feature. Even for simple features, the total number of scenarios can be staggering, thus making it impractical to document all the scenarios. This results in the designer just focusing on the major scenarios in the execution of a feature. This lack of focus on other scenarios causes lot of system instability; often leading to unhealthy finger pointing sessions after the system fails to perform optimally.
FDL solves the problems with multiple scenario definition by allowing designers to specify multiple scenarios in a single FDL file. Thus instead of defining multiple documents for each scenario, the designers just have to define multiple scenarios in a single file. In this section we will be covering how a single FDL file can store definitions of multiple scenarios. This is accomplished using the Case statement.

Case statements are used to specify flow of the feature in different scenarios. The leg statements in the case statement correspond to different scenarios. Case statements can be nested; this allows definition of really complex scenarios. Once the Case statements have been defined in an FDL file, the Scenario Wizard is used to define multiple scenarios by selecting the case legs to be taken in each scenario. If this seems a bit complicated, look at the example below to understand how scenarios are defined.

If-statements in FDL build upon the case statements. The if-statements are useful when a single scenario selection leads to flow changes at multiple places. The if-statements can use the scenario selection from a case statement to choose between two different flows.

```fml
module : customer1:customer, exchange, customer2:customer
component : phone1:phone in customer1, phone2:phone in customer2
component : frontend1:frontend in exchange, core in exchange
component : frontend2: frontend in exchange
 eternal : call_mgr1:call_mgr in frontend1
 eternal : call_mgr2:call_mgr in frontend2
dynamic : call1:call in frontend1, call2:call in frontend2
/* Originating Call Setup */
feature "Call Setup"
  pick_up_phone : env_l -> phone1
  offhook : phone1 -> call_mgr1
  (* Subscriber lifts the phone to initiate a call *)
call_mgr1 takes action "Check Customer Privileges"
call_mgr1 allocates "Subscriber Terminal"
call_mgr1 creates call1
  call1 state = "Idle"
  offhook : call_mgr1 -> call1
  call1 begins action "Feeding Dialtone"
  dialtone : call1 -> phone1
  call1 state = "Awaiting Digits"
  call1 starts await_first_digit_timer
  dial_digits : env_l -> phone1
  digits : phone1 -> call1
  call1 ends action "Feeding Dialtone"
call1 stops await_first_digit_timer
call1 starts more_digits_timer
  timeout more_digits_timer
  setup_call(digits="1-800-433-444", mode=NORMAL) : call1 -> core
  call1 state = "Awaiting Call Setup"
call_setup_request : core -> call_mgr2
call_mgr2 creates call2
call_setup_request : call_mgr2 -> call2
  ring_phone : call2 -> phone2
  ringing : phone2 -> env_r
  call2 state = "Ringing Subscriber"
call_setup_response : call2 -> core
```
setup_complete : core -> call1
call state = "Awaiting Answer"
[* Setup completed, waiting for called subscriber to be rung *]
ringback : core -> phone1

1. case
2. leg "called subscriber answers the call":
pick_up_phone : env_r -> phone2
answer : phone2 -> call2
answer_received: call2 -> call1
[* Call is now in conversation mode *]
3. case
4. leg "called subscriber hangs up first":
   hang_up : env_r -> phone2
   Clear_back : phone2 -> call2
   Clear_back : call2 -> call1
   leg "calling subscriber hangs up first":
   endcase
    endcase
4. leg "Called subscriber does not answer the call":
   leg "No answer and subscriber has voice mail service":
   issue "Voice mail call handling needs to be added"
    endcase
5. if "called subscriber answers the call"
   LogCall(status=METERED_CALL) : call2 -> core
6. else
   LogCall(status=UNMETERED_CALL) : call2 -> core
   stop_ring : call2 -> phone2
    endif
7. pagebreak
   released_call : call2 -> call1_mgr2
   call1_mgr2 deletes call2
   released_call : call1_mgr2 -> call1
   call_over : call1 -> call1_mgr1
   call1_mgr1 deletes call1
   call1_mgr1 frees "Subscriber Terminal"
endfeature

1. The case statement is enclosed in the case and endcase keywords. The case statement defines different legs that can be taken by the feature at this point. When EventStudio encounters a Case statement, it expects an input on which leg of the case statement has to be selected. This input is usually provided by the Scenario definition in a Scenario Project.

2. This statement declares the first leg in the case statement. This leg defines the feature leg taken when the subscriber answers the call. The statements that follow a leg statement will be executed only if the leg has been selected.

3. Any FDL statement can be used after the leg statement. Here we have shown a nested case statement. This case statement specifies the differences in flow, depending on which subscriber clears the call first.

4. This is the next leg for the outer case statement. This also marks the end of flow for the leg statement described in step (2.). Note that this leg statement does not have any FDL statements specified after it. This means that for this leg no specific action is required.

5. This statement demonstrates the if-else-endif statement. The if-statement switches on a previously defined leg label. The if-leg will be taken if the corresponding case leg was selected in the case statement. The else leg will be taken if the corresponding leg was not selected in the case statement.
6. The `else-endif` block is specified here. This block is optional, i.e. an `if-endif` statement might also be used, if no specific flow needs to be specified in the else leg.

7. Here we show the `pagebreak` statement. This statement inserts a page break in a PDF file. Use this statement whenever you wish to start the output on a new page.

**Explore More**

- **Multiple Scenario Modeling**
VISUAL GUIDE TO FDL

This section provides a visual introduction to FDL. Sequence diagrams and the associated FDL are shown.

Architecture Definition

```
#include "inc.fdl"

system: Earth
subsystem : "North America" in Earth, Asia in Earth
module: Canada in "North America", "United States" in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada, Montreal in Quebec, Jaipur in Rajasthan

feature "Action Statements"
  "San Francisco" takes action "Object Level Action"
  California takes action "Component Level Action"
  "United States" takes action "Module Level Action"
  "North America" takes action "Subsystem Level Action"
  Earth takes action "System Level Action"
endfeature
```

Explore More

Click on the links on the right for details.

- System declaration
- Subsystem declaration
- Module declaration
- Component declaration
- Eternal Object declaration
- Dynamic Object declaration
**Messages**

<table>
<thead>
<tr>
<th>Message Statements (Samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module 01</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>1: Message</td>
</tr>
<tr>
<td>2: Message Reply</td>
</tr>
<tr>
<td>3: Message With Parameters</td>
</tr>
<tr>
<td>param1 = &quot;Value 1&quot;, param2</td>
</tr>
<tr>
<td>4: Bold Message</td>
</tr>
<tr>
<td>5: Bidirectional</td>
</tr>
<tr>
<td>6: Chain 1</td>
</tr>
<tr>
<td>7: Chain 2</td>
</tr>
<tr>
<td>8: Multicast Message</td>
</tr>
<tr>
<td>9: Self Message</td>
</tr>
<tr>
<td>10: Lost Message</td>
</tr>
<tr>
<td>11: Message To Right Environment</td>
</tr>
</tbody>
</table>

**module**: Module_01  
**component**: Component_01 in Module_01  
**eternal**: a in Component_01, b in Component_01, c in Component_01

**feature** "Message Statements"

- Message : a -> b (* A simple message exchange *)
- "Message Reply": a <- b (* Reply to the message. *)
- "Message With Parameters" (param1="Value 1", param2) : b -> c (* Parameters can be specified with messages. *)
- "Bold Message" : b <= c (* Message shown with a thick arrow. *)
- "Bidirectional" : a <-> b (* Model bi-directional interactions. *)

chain

- "Chain 1": a -> b
- "Chain 2": b -> c

endchain (* Messages can be chained to appear in a single line. *)

- b multicasts "Multicast Message" to a, c (* Model multicasts. *)
- "Self Message" : b -> b (* A message sent to self. *)
- "Lost Message" : a ->X b (* Model a lost message. *)
- "Message To Right Environment": a -> env_r (* External entities can be modeled as environment. *)

endfeature

Explore More

- Message statement
- Chain statement
- Multicast statement
Message Tagging

```
eternal: a, b, c
feature "Message Statements"
  "Message Cascade": a-sip \rightarrow b-udp \rightarrow c-sdp
  (* Multiple tagged interactions in a single line. *)
  "Message Reply": a-sip \leftarrow b-udp \leftarrow c-sdp
  (* Tagged message cascade reply. *)
  "Bold Message": b-udp \rightarrow c
  (* Destination tagged Message shown with a thick arrow. *)
  "Bidirectional": a-sip \leftrightarrow b-q931
  (* Tagged bi-directional interactions. *)
  b-udp multicasts "Multicast Message" to a-udp, c-udp
  (* Tagged multicast. *)
  "Self Message": b-sip \rightarrow b-sdp
  (* A message sent to self. *)
  "Lost Message": a-sip \rightarrow X b-udp
  (* Model a lost message. *)
endfeature
```

Explore More
Click on the links on the right for details
- Message statement
- Chain statement
- Multicast statement
Object Interactions

module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Object Interactions"
  a invokes b.PerformInteraction(par1 = Value, par2, par3 = "Val 3")
  (* a invokes the PerformInteraction method supported by b. *)
  b takes action "Check parameter values"
  b invokes c.DelegateWork
  (* b further invokes the method DelegateWork in c. *)
  c takes action "Perform the Work"
  c.DelegateWork returns
  (* The method c.DelegateWork returns back to the caller. *)
  b.PerformInteraction returns (status = SUCCESS)
  (* b.PerformInteraction returns success to the caller. *)
endfeature

Explore More
Click on the link on the right for details

Invokes and Returns statements
**Object Creation and Deletion**

---

**Create statement**
- Create statement
- Delete statement

---

```plaintext
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01
dynamic: b in Component_01, c1 | c2 in Component_01

/* Dynamic object c1 and c2 share an axis. Only one of them can be active at a given time */

feature "Object Creation and Deletion"
  a creates b
  /* Eternal object 'a' dynamically creates dynamic object 'b'. */
  b creates c1(par1, par2="Value 2")
  /* Dynamic object 'b' dynamically creates dynamic object 'c1'. */
  b deletes c1
  /* Dynamic object 'b' deletes dynamic object 'c1'. */
  a deletes b
  /* Eternal object 'a' deletes dynamic object 'b'. */
  create c2
  /* Anonymous create. Note that c2 reuses the axis used by c1. */
  delete c2
  /* Anonymous delete. */

endfeature
```
Timers

module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01

feature "Timer Statements"
  b starts Timer_01 (* Start timer. *)
  b restarts Timer_01 (* Restart the timer. *)
  b stops Timer_01 (* Stop the timer. *)
  b starts Timer_02 (* Start timer. *)
  timeout Timer_02 (* Timer has expired. *)
  b starts periodic Timer_03
    (* Starting a periodic timer. The timer gets automatically Restarted on timer expiry. *)
  timeout Timer_03
  timeout Timer_03
  b stops Timer_03

endfeature

Explore More
Click on the links on the right for details
- Timer Start (One Shot and Periodic) statement
- Timer Restart statement
- Timer Stop statement
- Timeout statement
**Actions**

**Single Action**

```plaintext
#include "inc.fdl"

system: Earth
subsystem: "North America" in Earth, Asia in Earth
module: Canada in "North America", "United States" in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada, Montreal in Quebec, Jaipur in Rajasthan

feature "Action Statements"
  "San Francisco" takes action "Object Level Action"
  California takes action "Component Level Action"
  "United States" takes action "Module Level Action"
  "North America" takes action "Subsystem Level Action"
  Earth takes action "System Level Action"
endfeature
```

*Explore More*

Click on the links on the right for details.
Multiple Action

```plaintext
#include "inc.fdl"

system: Earth
subsystem: "North America" in Earth, Asia in Earth
module: Canada in "North America", "United States" in "North America"
module: India in Asia
component: California in "United States", Nevada in "United States"
component: Quebec in Canada, Rajasthan in India
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada, Montreal in Quebec, Jaipur in Rajasthan
feature "Multiple Action Statements"
   "San Francisco", "Las Vegas" take action "Object Level Joint Action"
   California, Quebec take action "Component Level Joint Action"
   "United States", Canada take action "Module Level Joint Action"
   "North America", Asia take action "Subsystem Level Joint Action"
   "United States", Jaipur take action "Module and Object Joint Action"
endfeature
```

Explore More
Click on the links on the right for details.

- Action statement
- Action Begin statement
- Action End statement
**Resource Allocations**

```
module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Resource Allocations"
  b allocates "Frequency" (* Allocate a resource. *)
  b frees "Frequency" (* Free a resource. *)
  b, c allocate "Call Resources" (* Joint resource allocate. *)
  b, c free "Call Resources" (* Joint resource free. *)
  Module_01 allocates "Module Resources" (* Module level resource allocation. *)
  Module_01 frees "Module Resources" (* Free module level resources. *)
endfeature
```

Explore More
Click on the links on the right for details.

- **Resource Allocate statement**
- **Resource Free statement**
State Transitions

module: Module_01, Module_02
component: Component_01 in Module_01, Component_02 in Module_02
eternal: a in Component_01, b in Component_01, c in Component_02
feature "State Transitions"
  b state = "Awaiting Dial-tone"  (* Individual object state. *)
  a,b,c state = "Conversation"  (* Joint state transition. *)
  Component_01 state = "Call Active"  (* Component level state transition. *)
endfeature

Explore More
Click on the link on the right for details.

State Change statement
Sequence Groupings

module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Sequence Grouping"
  sequence "Call Setup"
    IAM : a -> b
    b takes action "Check digits"
    ACM : a <- b
  endsequence
  case
    leg "Called Party Free":
      a, b take action "Set up the voice path"
    leg "Called Party Busy":
      a, b take action "Feed busy tone"
  endcase
  sequence "Call Release"
    REL : a -> b
    RLC : a <- b
  endsequence
endfeature

Explore More
Click on the link on the right for details.

- Sequence Block Statement
Remarks

module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Sequence Grouping"
  msg : a -> b /* A remark is printed on the right hand side. The remark is associated with the statement preceding it. */
  /* A block remark can be used to display detailed comments across the width of a sequence diagram. */
  /* Text can be split into paragraphs by using an extra line break. */
  /* C-style comment, ignored in document generation. */
  heading "Heading. Show up as bookmarks in PDF documents."
  issue "Results in a warning. Used to track unresolved issues"
endfeature

Explore More
Click on the link on the right for details.

- Remark statement
- Block Remark statement
- Comment statement
- Heading statement
- Issue statement
FDL REFERENCE

This chapter provides a comprehensive reference to FDL. The following topics are covered:

- Architecture and Feature Definition
- Multiple Scenario Modeling
- Object Interactions
- Object Creation and Deletion
- Timers
- Resource Allocations
- State Transitions
- Remarks
- Preprocessor
- Styles, Themes and Document Layout

Architecture and Feature Definition

System declaration

This statement declares the systems that will be used in the feature. All the systems that are being used in the subsystem declaration statements must be defined using a system declaration statement before it.

The order of the system declaration determines the order of the columns assigned to the systems in the Sequence Diagrams.

System names can be specified as:

- **Identifiers**: Identifiers can contain alphanumeric characters and an underscore. The name should begin only with an alpha character. Also, the underscore characters used in the name are treated as separators. If the column assigned to print the name is too small, the name will be split at the underscore boundaries and will be displayed in multiple rows.

- **Strings**: Strings are enclosed in "quotes". Blanks in the strings are treated as separators.

Systems are the highest level entities in five-tier decomposition. No system is defined in a four-tier design or three-tier architecture or two-tier architecture or single-tier architecture.

FIVE-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>System</th>
<th></th>
<th>Earth</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsystem</td>
<td></td>
<td>&quot;North America&quot;</td>
</tr>
<tr>
<td>module</td>
<td></td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>component</td>
<td></td>
<td>California</td>
</tr>
<tr>
<td>eternal</td>
<td></td>
<td>&quot;Los Angeles&quot;</td>
</tr>
</tbody>
</table>

```
| System : Earth |
| subsystem : "North America" in Earth, Asia in Earth |
| module : "United States" in "North America", Canada in "North America" |
| module : India in Asia |
```
**component**: California in "United States", Nevada in "United States"
**component**: Quebec in Canada, Rajasthan in India
**eternal**: "Los Angeles" in California, "San Francisco" in California
**eternal**: "Las Vegas" in Nevada
**eternal**: Montreal in Quebec, Jaipur in Rajasthan

**MORE EXAMPLES**

**system**: routing_area, msc
Two systems define a five-tier system architecture.

**system**: calling_party:customer, called_party:customer
Two systems calling_party and called_party have been declared. Further, it is declared that both have a type customer.

**system**: "HTTP Server", "web Browser"
String based system names.

**system**: [intranet_style] intranet, [internet_style] internet
Systems have been declared with a style specification. The font, line width, line style, color and image specified in the style are used to draw the system (For details refer to Style declaration).

**Notes**:  
- System instance style is used in the "System interaction sequence diagrams".
- It is recommended that light weight header (with or without image) be selected when system instance style is used (The light weight header can be selected from the "Tools->Options->Customize tab" menu selection).

**Subsystem declaration**

This statement declares the subsystems that will be used in the feature. All the subsystems that are being used in the module declaration statements must be defined using a subsystem declaration statement before it.

The order of the subsystem declaration determines the order of the columns assigned to the subsystems in the Sequence Diagrams.

Subsystem names can be specified as:

- **Identifiers**: Identifiers can contain alphanumeric characters and an underscore. The name should begin only with an alpha character. Also, the underscore characters used in the name are treated as separators. If the column assigned to print the name is too small, the name will be split at the underscore boundaries and will be displayed in multiple rows.

- **Strings**: Strings are enclosed in "quotes". Blanks in the strings are treated as separators.

Subsystems are contained in a system when five-tier decomposition is used. Subsystems are the highest level entities in four-tier decomposition. No subsystem is defined in a three-tier architecture, two-tier architecture or single-tier architecture.
FIVE-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>System</th>
<th>Earth</th>
<th>Asia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem</td>
<td>&quot;North America&quot;</td>
<td>Asia</td>
</tr>
<tr>
<td>Module</td>
<td>&quot;United States&quot;</td>
<td>Canada, India</td>
</tr>
<tr>
<td>Component</td>
<td>California</td>
<td>Nevada, Quebec, Rajasthan</td>
</tr>
<tr>
<td>Eternal</td>
<td>&quot;Los Angeles&quot;</td>
<td>&quot;San Francisco&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Las Vegas&quot;</td>
<td>Montreal, Jaipur</td>
</tr>
</tbody>
</table>

FOUR-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>&quot;North America&quot;</th>
<th>Asia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>&quot;United States&quot;</td>
<td>Canada, India</td>
</tr>
<tr>
<td>Component</td>
<td>California</td>
<td>Nevada, Quebec, Rajasthan</td>
</tr>
<tr>
<td>Eternal</td>
<td>&quot;Los Angeles&quot;</td>
<td>&quot;San Francisco&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Las Vegas&quot;</td>
<td>Montreal, Jaipur</td>
</tr>
</tbody>
</table>

MORE EXAMPLES

- **subsystem : routing_area, msc**
  Two subsystems define a four-tier system architecture. A four-tier architecture is used as the subsystems are not contained within a system.

- **subsystem : calling_party:customer, called_party:customer**
  Two subsystems calling_party and called_party have been declared. Further, it is declared that both have a type customer.

- **subsystem : "HTTP Server" in Internet, "Web Browser" in Internet**
  String based subsystem names. The two subsystems are contained in the Internet system.

- **subsystem : [intranet_style] intranet, [internet_style] internet**
  Subsystems have been declared with a style specification. The font, line width, line style, color and image specified in the style are used to draw the subsystem (For details refer to section 0 - Style declaration).

- **subsystem : "United States" in "North America", Ghana in Africa**
  This example shows subsystems included in a system. The declaration states that the subsystem "United States" is contained in the system "North America" and Ghana subsystem is contained in the Africa system.

**Notes:**
- Subsystem instance style is used in the "subsystem interaction sequence diagrams".
• It is recommended that light weight header (with or without image) be selected when subsystem instance style is used (The light weight header can be selected from the "Tools->Options->Customize tab" menu selection).

Module declaration
This statement declares the modules that will be used in the feature. All the modules that are being used in the component declaration statements must be defined using the module declaration statement before it.

The order of the module declaration determines the order of the columns assigned to the modules in the Sequence Diagrams. The modules are displayed at the top of the Sequence Diagrams along the full output page width, immediately below the feature title in three-tier decomposition header.

Module names can be specified as:

• **Identifiers**: Identifiers can contain alphanumeric characters and an underscore. The name should begin only with an alpha character. Also, the underscore characters used in the name are treated as separators. If the column assigned to print the name is too small, the name will be split at the underscore boundaries and will be displayed in multiple rows.

• **Strings**: Strings are enclosed in "quotes". Blanks in the strings are treated as separators.

Modules are contained in a subsystem in four-tier or five-tier decomposition. Modules are the highest level entities in three-tier decomposition. No module is defined in two-tier decomposition or single-tier architecture.

### FIVE-TIER DECOMPOSITION EXAMPLE

| system | Earth |
| subsystem | "North America" |
| module | "United States" |
| component | California |
| eternal | "Los Angeles", "San Francisco", "Las Vegas" |

```
system : Earth
subsystem : "North America" in Earth, Asia in Earth
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
permanent : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan
```

### FOUR-TIER DECOMPOSITION EXAMPLE

| subsystem | "North America" |
| module | "United States" |
| component | California |
| eternal | "Los Angeles", "San Francisco", "Las Vegas" |

```
subsystem : "North America", Asia
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
```
eternal: "Los Angeles" in California, "San Francisco" in California
eternal: "Las Vegas" in Nevada
eternal: Montreal in Quebec, Jaipur in Rajasthan

THREE-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>Module</th>
<th>United States</th>
<th>Canada</th>
<th>India</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>California</td>
<td>Nevada</td>
<td>Quebec</td>
</tr>
<tr>
<td>Eternal</td>
<td>&quot;Los Angeles&quot;</td>
<td>&quot;San Francisco&quot;</td>
<td>&quot;Las Vegas&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>United States, Canada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>India in Asia</td>
</tr>
<tr>
<td>Component</td>
<td>California in &quot;United States&quot;, Nevada in &quot;United States&quot;</td>
</tr>
<tr>
<td>Component</td>
<td>Quebec in Canada, Rajasthan in India</td>
</tr>
<tr>
<td>Eternal</td>
<td>&quot;Los Angeles&quot; in California, &quot;San Francisco&quot; in California</td>
</tr>
<tr>
<td>Eternal</td>
<td>&quot;Las Vegas&quot; in Nevada</td>
</tr>
<tr>
<td>Eternal</td>
<td>Montreal in Quebec, Jaipur in Rajasthan</td>
</tr>
</tbody>
</table>

MORE EXAMPLES

module: customer, exchange
Here, two modules customer and exchange have been declared.

module: calling_party:customer, called_party:customer
Here, two modules calling_party and called_party have been declared. Further, it is declared that both have a type customer.

module: "HTTP Server", "Web Browser"
String based module names.

module: [intranet_style] intranet, [internet_style] internet
Modules have been declared with a style specification. The font, line width, line style, color and image specified in the style are used to draw the modules (For details refer to section 0 - Style declaration).

module: "United States" in "North America", Ghana in Africa
This example shows modules included in a subsystem. The declaration states that the module "United States" is contained in the subsystem "North America" and Ghana module is contained in the Africa subsystem.

Notes:
- Module instance style is used in the "module interaction sequence diagrams".
- It is recommended that light weight header (with or without image) be selected when module instance style is used (The light weight header can be selected from the "Tools->Options->Customize tab" menu selection).

Component declaration
This statement declares components that will be used in the feature. All the components that are being used in the eternal or dynamic declaration statements must be defined using the component declaration statement before it. A component declaration statement can be put only inside the FDL declaration block at the beginning of the FDL file. There can be no component declaration statement inside the feature-endfeature block in the FDL file.

The order of the component declaration determines the order of the columns assigned to the components in the Sequence Diagrams.
Component names can be specified as:

- **Identifiers**: Identifiers can contain alphanumeric characters and an underscore. The name should begin only with an alpha character. Also, the underscore characters used in the name are treated as separators. If the column assigned to print the name is too small, the name will be split at the underscore boundaries and will be displayed in multiple rows.

- **Strings**: Strings are enclosed in "quotes". Blanks in the strings are treated as separators.

### FIVE-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>system</th>
<th>Earth</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsystem</td>
<td>&quot;North America&quot;</td>
</tr>
<tr>
<td>module</td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>component</td>
<td>California</td>
</tr>
<tr>
<td>eternal</td>
<td>&quot;Los Angeles&quot;, &quot;San Francisco&quot;, &quot;Las Vegas&quot;</td>
</tr>
</tbody>
</table>

FIVE-TIER DECOMPOSITION EXAMPLE

```
system : Earth
subsystem : "North America" in Earth, Asia in Earth
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan
```

### FOUR-TIER DECOMPOSITION EXAMPLE

| subsystem | "North America" |
| module | "United States" |
| component | California |
| eternal | "Los Angeles", "San Francisco", "Las Vegas" |

FOUR-TIER DECOMPOSITION EXAMPLE

```
subsystem : "North America", Asia
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan
```

### THREE-TIER DECOMPOSITION EXAMPLE

| module | "United States" |
| component | California |
| eternal | "Los Angeles", "San Francisco", "Las Vegas" |

THREE-TIER DECOMPOSITION EXAMPLE

```
module : "United States", Canada
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan
```
TWO-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>Component</th>
<th>California</th>
<th>Nevada</th>
<th>Quebec</th>
<th>Rajasthan</th>
</tr>
</thead>
<tbody>
<tr>
<td>eternal</td>
<td>&quot;Los Angeles&quot;</td>
<td>&quot;San Francisco&quot;</td>
<td>&quot;Las Vegas&quot;</td>
<td>Montreal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jaipur</td>
</tr>
</tbody>
</table>

MORE EXAMPLES

**component**: California, Nevada
**component**: Quebec, Rajasthan
**eternal**: "Los Angeles" in California, "San Francisco" in California
**eternal**: "Las Vegas" in Nevada
**eternal**: Montreal in Quebec, Jaipur in Rajasthan

Notes:
- Component instance style is used in the "component interaction sequence diagrams".
- It is recommended that light weight header (with or without image) be selected when component instance style is used (The light weight header can be selected from the "Tools->Options->Customize tab" menu selection).

Eternal Object declaration

This statement is used to declare an object that is created at system startup and exists throughout the life of the system. An eternal object declaration statement can be put only inside the FDL declaration block at the beginning of the FDL file.

The order of the object declaration statements determines the order of the columns assigned to the objects in the Sequence Diagrams. The instance axis of an eternal object is displayed from the beginning of the document to its end.

Eternal object names can be specified as:

- **Identifiers**: Identifiers can contain alphanumeric characters and an underscore. The name should begin only with an alpha character. Also, the underscore characters used in the name are treated as separators. If the column assigned to print the name is too small, the name will be split at the underscore boundaries and will be displayed in multiple rows.
• **Strings:** Strings are enclosed in "quotes". Blanks in the strings are treated as separators.

**FIVE-TIER DECOMPOSITION EXAMPLE**

<table>
<thead>
<tr>
<th>system</th>
<th>Earth</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsystem</td>
<td>&quot;North America&quot;</td>
</tr>
<tr>
<td>module</td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>component</td>
<td>California, Nevada, Quebec, Rajasthan</td>
</tr>
<tr>
<td>eternal</td>
<td>&quot;Los Angeles&quot;, &quot;San Francisco&quot;, &quot;Las Vegas&quot;, Montreal, Jaipur</td>
</tr>
</tbody>
</table>

system : Earth
subsystem : "North America" in Earth, Asia in Earth
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan

**FOUR-TIER DECOMPOSITION EXAMPLE**

<table>
<thead>
<tr>
<th>subsystem</th>
<th>&quot;North America&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>component</td>
<td>California, Nevada, Quebec, Rajasthan</td>
</tr>
<tr>
<td>eternal</td>
<td>&quot;Los Angeles&quot;, &quot;San Francisco&quot;, &quot;Las Vegas&quot;, Montreal, Jaipur</td>
</tr>
</tbody>
</table>

subsystem : "North America", Asia
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan

**THREE-TIER DECOMPOSITION EXAMPLE**

<table>
<thead>
<tr>
<th>module</th>
<th>&quot;United States&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>California, Nevada, Quebec, Rajasthan</td>
</tr>
<tr>
<td>eternal</td>
<td>&quot;Los Angeles&quot;, &quot;San Francisco&quot;, &quot;Las Vegas&quot;, Montreal, Jaipur</td>
</tr>
</tbody>
</table>

module : "United States", Canada
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan

**TWO-TIER DECOMPOSITION EXAMPLE**

| component | California, Nevada, Quebec, Rajasthan |
| eternal | "Los Angeles", "San Francisco", "Las Vegas", Montreal, Jaipur |
component : California, Nevada
component : Quebec, Rajasthan
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan

SINGLE-TIER DECOMPOSITION EXAMPLE

| eternal | "Los Angeles" | "San Francisco" | "Las Vegas" | Montreal | Jaipur |

| eternal | "Los Angeles", "San Francisco" |
| eternal | "Las Vegas" |
| eternal | Montreal, Jaipur |

MORE EXAMPLES

eternal : call_mgr
Here, an eternal object call_mgr is declared at the topmost level in two-tier decomposition architecture.

eternal : call_mgr in frontend
Here, an eternal object call_mgr is declared inside the frontend component.

eternal : call_mgr1:call_mgr in frontend1
eternal : call_mgr2:call_mgr in frontend2
Here, eternal objects call_mgr1 and call_mgr2 are declared. The call_mgr1 object has been specified inside the component frontend1 and call_mgr2 in frontend2. Further, call_mgr1 and call_mgr2 have been declared to be of object type call_mgr.

eternal : "I-CSCF" in IMS, "S-CSCF" in IMS
Eternal objects are declared as strings.

eternal : [object_style] sip_call_manager in server
Eternal objects have been declared with a style specification. The font, line width, line style, color and image specified in the style are used to draw the eternal objects (For details refer section 0 - Style declaration).

Notes:  
- It is recommended that light weight header (with or without image) be selected when eternal object instance style is used (The light weight header can be selected from the "Tools->Options->Customize tab" menu selection).

Dynamic Object declaration
This statement is used to declare an object that will be dynamically created and deleted by the system. A dynamic object declaration statement can be put only inside the FDL declaration block at the beginning of the FDL file. Use the create statement to create a dynamic object. Use the delete statement when a dynamic object needs to be deleted. A dynamic object created in a feature must be deleted before the feature ends.
The order of the object declaration statements determines the order of the columns assigned to the objects in the Sequence Diagrams. The instance axis of a dynamic object is displayed from the creation time to the deletion time.

One instance axis is assigned to a dynamic object. To conserve space you can share a single instance axis between multiple dynamic objects. See the Advanced Examples for details.

Dynamic object names can be specified as:

- **Identifiers:** Identifiers can contain alphanumeric characters and an underscore. The name should begin only with an alpha character. Also, the underscore characters used in the name are treated as separators. If the column assigned to print the name is too small, the name will be split at the underscore boundaries and will be displayed in multiple rows.

- **Strings:** Strings are enclosed in "quotes". Blanks in the strings are treated as separators.

**FIVE-TIER DECOMPOSITION EXAMPLE**

<table>
<thead>
<tr>
<th>system</th>
<th>Earth</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsystem</td>
<td>&quot;North America&quot;</td>
</tr>
<tr>
<td>module</td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>component</td>
<td>California</td>
</tr>
<tr>
<td>dynamic</td>
<td>&quot;Los Angeles&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;San Francisco&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Las Vegas&quot;</td>
</tr>
<tr>
<td></td>
<td>Montreal</td>
</tr>
<tr>
<td></td>
<td>Jaipur</td>
</tr>
</tbody>
</table>

**FOUR-TIER DECOMPOSITION EXAMPLE**

<table>
<thead>
<tr>
<th>subsystem</th>
<th>&quot;North America&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>&quot;United States&quot;</td>
</tr>
<tr>
<td>component</td>
<td>California</td>
</tr>
<tr>
<td>dynamic</td>
<td>&quot;Los Angeles&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;San Francisco&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;Las Vegas&quot;</td>
</tr>
<tr>
<td></td>
<td>Montreal</td>
</tr>
<tr>
<td></td>
<td>Jaipur</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>system</th>
<th>Earth</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsystem</td>
<td>&quot;North America&quot; in Earth, Asia in Earth</td>
</tr>
<tr>
<td>module</td>
<td>&quot;United States&quot; in &quot;North America&quot;, Canada in &quot;North America&quot;</td>
</tr>
<tr>
<td>component</td>
<td>California in &quot;United States&quot;, Nevada in &quot;United States&quot;</td>
</tr>
<tr>
<td>dynamic</td>
<td>&quot;Los Angeles&quot; in California, &quot;San Francisco&quot; in California</td>
</tr>
<tr>
<td></td>
<td>&quot;Las Vegas&quot; in Nevada</td>
</tr>
<tr>
<td></td>
<td>Montreal in Quebec, Jaipur in Rajasthan</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>subsystem</th>
<th>&quot;North America&quot;, Asia</th>
</tr>
</thead>
<tbody>
<tr>
<td>module</td>
<td>&quot;United States&quot; in &quot;North America&quot;, Canada in &quot;North America&quot;</td>
</tr>
<tr>
<td>component</td>
<td>California in &quot;United States&quot;, Nevada in &quot;United States&quot;</td>
</tr>
<tr>
<td>dynamic</td>
<td>&quot;Los Angeles&quot; in California, &quot;San Francisco&quot; in California</td>
</tr>
<tr>
<td></td>
<td>&quot;Las Vegas&quot; in Nevada</td>
</tr>
<tr>
<td></td>
<td>Montreal in Quebec, Jaipur in Rajasthan</td>
</tr>
</tbody>
</table>
THREE-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>module</th>
<th>United States</th>
<th>Canada</th>
<th>India</th>
</tr>
</thead>
<tbody>
<tr>
<td>component</td>
<td>California</td>
<td>Nevada</td>
<td>Quebec</td>
</tr>
<tr>
<td>dynamic</td>
<td>&quot;Los Angeles&quot;</td>
<td>&quot;San Francisco&quot;</td>
<td>&quot;Las Vegas&quot;</td>
</tr>
</tbody>
</table>

module : "United States", Canada
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
dynamic : "Los Angeles" in California, "San Francisco" in California
dynamic : "Las Vegas" in Nevada
dynamic : Montreal in Quebec, Jaipur in Rajasthan

TWO-TIER DECOMPOSITION EXAMPLE

<table>
<thead>
<tr>
<th>component</th>
<th>California</th>
<th>Nevada</th>
<th>Quebec</th>
<th>Rajasthan</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>&quot;Los Angeles&quot;</td>
<td>&quot;San Francisco&quot;</td>
<td>&quot;Las Vegas&quot;</td>
<td>Montreal</td>
</tr>
</tbody>
</table>

component : California, Nevada
component : Quebec, Rajasthan
dynamic : "Los Angeles" in California, "San Francisco" in California
dynamic : "Las Vegas" in Nevada
dynamic : Montreal in Quebec, Jaipur in Rajasthan

SINGLE-TIER DECOMPOSITION EXAMPLE

dynamic : "Los Angeles", "San Francisco", "Las Vegas", Montreal, Jaipur

dynamic : "Los Angeles", "San Francisco"
dynamic : "Las Vegas"
dynamic : Montreal, Jaipur

MORE EXAMPLES

dynamic : call
Here, a dynamic object call is defined. It happens to be at the topmost level in single-tier decomposition architecture.

dynamic : call1:call in frontend, call2:call in frontend2
Here, dynamic objects call1 and call2 are declared. The call1 object has been declared inside the component frontend1 and call2 inside frontend2. Further, call1 and call2 have been declared to be of object type call.
Dynamic objects declared as strings.

ADVANCED EXAMPLES

**dynamic :** "Terminating Call" in "SIP Server"

This is an example of two dynamic objects sharing the same axis. Only one of these objects can be active at a time. Organizing dynamic objects this way is useful when different scenarios instantiate different objects and dedicating a separate instance axis for each object would lead to space wastage.

**dynamic :** ISUPCall | SIPCall in "Signaling Gateway"

Eternal objects have been declared with a style specification. The font, line width, line style, color and image specified in the style are used to draw the dynamic objects (For details refer to section 0 - **Style declaration**).

**Notes:**

- It is recommended that light weight header (with or without image) be selected when dynamic object instance style is used (The light weight header can be selected from the "Tools->Options->Customize tab" menu selection).

**Type declaration**

FDL allows defining types for a system, subsystem, module, component, eternal object or dynamic object. This is used to specify that two or more different entities are really instances of the same entity type. A type declaration statement can be put only inside the FDL declaration block at the beginning of the FDL file. There can be no type declaration statement inside the feature-endfeature block in the FDL file.

It may be noted that defining the types will have no visual impact on the PDF Sequence Diagrams. The main difference can be seen in the definition of Unit Test Procedures and Interface documents. If the user specifies the type for generating a document, EventStudio will include interfaces of all instances of the given type.

**EXAMPLES**

**module :** calling_party:customer, called_party:customer

Here, two modules calling_party and called_party have been declared. Further, it is declared that both have a type customer. If the user specifies customer for generating an Interface document, the interfaces of calling_party and called_party will be included. The user still has the option of generating Interface documents for individual modules by specifying called_party or calling_party as the filter.

**component :** "P-CSCF 1":"P-CSCF" in IMS, "P-CSCF 1":"P-CSCF" in IMS

Declares "P-CSCF 1" and "P-CSCF 2" components, both the components are in the IMS module. Further "P-CSCF 1" and "P-CSCF 2" are declared to be of the type "P-CSCF". If the user specifies "P-CSCF" for generating an Interface document, the interfaces of "P-CSCF 1" and "P-CSCF 2" will be included. The user still has the option of generating Interface documents for individual components by specifying "P-CSCF 1" or "P-CSCF 2" as the filter.

**eternal :** call_mgr1:call_mgr in frontend1
**eternal :** call_mgr2:call_mgr in frontend2

Here, eternal objects call_mgr1 and call_mgr2 are declared. The call_mgr1 object has been specified inside the component frontend1 and call_mgr2 in frontend2. Further, call_mgr1 and call_mgr2 have been declared to be of object type call_mgr. If the user specifies call_mgr for generating an Interface document, the interfaces of call_mgr1 and call_mgr2 will be included. The user still has the option of generating Interface
documents for individual components by specifying call_mgr1 or call_mgr2 as the filter. In a similar fashion, the user can generate Unit Test Procedures for call_mgr. This would include test cases for call_mgr1 and call_mgr2.

**dynamic : call1:call_in_frontend1, call2:call_in_frontend2**

Here, dynamic objects call1 and call2 are declared. The call1 object has been declared inside the component frontend1 and call2 inside frontend2. Further, call1 and call2 have been declared to be of object type call. If the user specifies call for generating an Interface document, the interfaces of call1 and call2 will be included. The user still has the option of generating Interface documents for individual objects by specifying call1 or call2 as the filter. In a similar fashion, the user can generate Unit Test Procedures for call. This would include test cases for call1 and call2.

**Environment declaration**

FDL allows the user to depict message interactions with the external world. This can be done by sending messages to the left environment (env_l) or right environment (env_r). As the names suggest, env_l and env_r are assumed to be at the left and right edges of the document. Environment interactions can be very useful in clearly spelling out what is external to the system being designed.

env_l and env_r are declared internally, thus there is no declaration statement for environments. env_l and env_r can be used in the message statement as source or destination of messages.

**EXAMPLES**

```
pick_up_phone : env_l -> phone
```

Here, a pick_up_phone message is received from the left environment. This will be depicted as an arrow from the left edge of the document to the axis corresponding to phone.

```
call_setup_request : core -> env_r
```

Core sends a message to the right environment. This will be depicted as an arrow from the core axis to the right environment.

**Feature statement**

This statement encloses all the message interactions for a feature. The **feature-endfeature** block follows the declarations in FDL file. A title for the feature should be specified in the feature declaration. A FDL file defines feature interactions for one feature. Hence there can be one feature-endfeature block in a FDL file.

A theme may be associated with a feature-endfeature block. A theme specifies the layout options and the fonts.

The feature title that is specified in the statement is displayed at the top, along the complete width of the output page of the Sequence Diagram.

EventStudio performs various checks on the feature-endfeature block. It issues a warning if it finds that timers have been left running at feature end. A warning is issued if a dynamic object is not deleted before the feature ends. A warning is issued if resources have not been deallocated at feature end.

**Note:** The strings used in this statement should not contain any leading or trailing blanks. Also words within the string should not be separated by more than one blank.

**EXAMPLES**
This feature-endfeature block encloses all the feature interactions for the call setup feature.

Here a feature-endfeature block is shown with a theme specification for the feature. The theme specifies the layout options and the font sizes.

**Multiple Scenario Modeling**

**Case statement**

This statement is used to specify multiple scenarios in the same FDL file. A case statement is enclosed within a case-endcase block. The legs for the case are included within the case-endcase block. Use the leg statement to define each scenario option. Nesting of case statements is also supported. A scenario determines the flow of a feature under certain conditions. To define a scenario, use the Scenario Wizard and select a leg from each case statement that is encountered in the feature flow. A case statement can be put only inside the feature-endfeature block in the FDL file.

There is no output corresponding to the case statement. The leg selected for defining the scenario is displayed in bold in the right side remarks column in a sequence diagram.

**EXAMPLES**

``` событие "Call Setup"
   msg1 : phone -> call_mgr
   msg2 : call_mgr -> core
endсобытие
```

Here a feature-endfeature block is shown with a theme specification for the feature. The theme specifies the layout options and the font sizes.

```MyTheme} событие "Call Setup"
   msg1 : phone -> call_mgr
   msg2 : call_mgr -> core
endсобытие
```

This feature-endfeature block encloses all the feature interactions for the call setup feature.
Here, a simple case statement is shown where three legs have been specified. As a part of scenario definition, when Scenario Wizard encounters this case statement, it prompts the user to select a leg from the three options.

```
leg "Switchover succeeds"

leg "Switchover aborted"

endcase
```

Here, a nested case statement is shown. As a part of scenario definition, when Scenario Wizard encounters this case statement, it prompts the user to select a leg from the outermost case statement. If the user selects "Partial dialing", the scenario definition would complete. If, however, the user selects "Complete dialing", Scenario Wizard will prompt the user to select another leg from the next nested case. If the user selects "Routing successful", the user is again prompted to select a leg from the innermost case statement. If the user selects "Successful call", the scenario definition for a successful call would complete.

**RELATION WITH IF-ELSE-ENDIF STATEMENT**

- Any case leg traversed in the case-endcase may be used in an if-else-endif or if-endif statement.
- Case-endcase statements may be enclosed within an "if" or "else" part of an if-else-endif or if-endif statements.

**Leg statement**

This statement is used to specify legs within a case-endcase block. The legs for the case are included within the case-endcase block. The leg statement defines each scenario option. A scenario determines the flow of a feature under certain conditions. To define a scenario, use the Scenario Wizard and select a leg from each case statement that is encountered in the feature flow. A leg statement can be put only inside the case-endcase block in the FDL.

The leg selected for defining the scenario is displayed in bold in the right side remarks column in a sequence diagram. The leg selections are also available as bookmarks in PDF sequence diagrams. When a PDF
sequence diagram is open, the leg selection bookmarks are shown on the left side of the screen. Clicking on
the bookmarks will take you directly to the leg selection.

**Note:** The strings used in this statement should not contain any leading or trailing blanks. Also words within the
string should not be separated by more than one blank.

**EXAMPLES**

```plaintext
case
  leg "Switchover fails":
  leg "Switchover succeeds":
  leg "Switchover aborted":
endcase
```

Here, a simple case statement is shown where three legs have been specified. As a part of scenario definition,
when Scenario Wizard encounters this case statement, it prompts the user to select a leg from the three
options.

```plaintext
case
  leg "Partial dialing":
  leg "Complete dialing":
    case
      leg "Routing failure":
      leg "Routing successful":
    endcase
    case
      leg "Outgoing trunk congestion":
      leg "Called party busy":
      leg "No answer":
      leg "Successful call":
    endcase
endcase
```

Here, a nested case statement is shown. As a part of scenario definition, when Scenario Wizard encounters
this case statement, it prompts the user to select a leg from the outermost case statement. If the user selects
"Partial dialing", the scenario definition would complete. If, however, the user selects "Complete dialing",
Scenario Wizard will prompt the user to select another leg from the next nested case. If the user selects
"Routing successful", the user is again prompted to select a leg from the innermost case statement. If the user
selects "Successful call", the scenario definition for a successful call would complete.

**If-Else-Endif statement**

This statement is used to choose different feature flows from previously encountered leg statements. The string
for the if-statement must have been defined as a leg of a previous case statement. When EventStudio
encounters the If-Else-Endif statement, it transfers control to the "If block" if the leg corresponding to the "if
string" has been chosen in a previous case statement. The control is transferred to the "Else block" if the leg corresponding to the "if string" has not been chosen. Look at the example given below to completely understand how this statement is used.

There is no output produced corresponding to this statement.

**EXAMPLES**

```
case
  leg "DTMF dialing":
  leg "Pulse dialing":
endcase

if "DTMF dialing"
  call frees "DTMF Receiver"
  call takes action "Increment DTMF counter"
else
  call takes action "Increment Pulse counter"
endif
```

Here, a case statement defines the legs "DTMF dialing" and "Pulse dialing". Later on in the feature flow if statement checks if "DTMF dialing" leg was chosen. If "DTMF dialing" leg was taken, control is transferred to the "if block". Otherwise the control is transferred to the "else block". It may be noted that EventStudio remembers the leg selections and user is not prompted for an input when an "if statement" is encountered.

```
case
  leg "Priority Delivery":
  leg "Standard Delivery":
endcase

case
  leg "Customer Present":
  leg "Customer Not Present":
endcase

if "Customer Not Present"
  if "Priority Delivery"
    driver takes action "Call the customer to reschedule delivery"
  else
    driver takes action "Leave the package at the door"
  endif
else
  call takes action "Deliver the package"
endif
```

The above example shows how an if-else-endif can be nested within another if-else-endif statement. If-else-endif and if-else statements can be nested to specify the handling for combination of leg statements.

**If-Endif statement**

This statement is used to choose different feature flows from previously encountered leg statements. The string for the "if statement" must have been defined as a leg of a previously defined case statement. When
EventStudio encounters the If-Endif statement, it transfers control to the "If block" if the leg corresponding to the "if string" has been chosen in a previous case statement. Look at the example given below to completely understand how this statement is used.

There is no output produced corresponding to this statement.

**EXAMPLE**
```
    case
      leg "DTMF dialing":
      leg "Pulse dialing":
      endcase

    if "DTMF dialing"
      call frees "DTMF Receiver"
    endif
```

Here, a case statement defines the legs "DTMF dialing" and "Pulse dialing". Later on in the feature flow if statement checks if "DTMF dialing" leg was chosen. If "DTMF dialing" case leg was taken, control is transferred to the "if block". Otherwise the control is transferred to the statement following "endif". It may be noted that EventStudio remembers the leg selections and user is not prompted for an input when an "if statement" is encountered.

```
    case
      leg "Priority Delivery":
      leg "Standard Delivery":
      endcase

    case
      leg "Customer Present":
      leg "Customer Not Present":
      endcase

    if "Customer Not Present"
      if "Priority Delivery"
        driver takes action "Call the customer to reschedule delivery"
      else
        driver takes action "Leave the package at the door"
      endif
    endif
```

The above example shows how an if-else-endif can be nested within a if-endif statement. If-else-endif and if-else statements can be nested to specify the handling for combination of leg statements.

**Goto and Label statements**

The Goto statement allows you to jump forward to a label defined at a later point in the FDL. Use this statement to exit the diagrams from error scenarios.

The following restrictions apply:
• A Goto statement can be used to jump forward only, i.e. jumping to a label defined before the goto statement is not allowed.
• A label cannot be defined within a case-endcase, if-endif or an if-else-endif block.

EXAMPLE

```
  case
    leg "No digits dialed":
      goto exit
    leg "All digits dialed":
  endcase

  case
    leg "Call routing successful":
    leg "Call routing failed":
      goto exit
  endcase

  label exit:
    ["call has ended. "]
```

A "goto" to the exit label is being used to end error scenarios in a call setup.

Messages

Message statement

This statement is used to represent message exchanges between the source and destination. The source and destination can be any of the following:

• Component (i.e. component without any other objects inside it)
• Eternal object
• Dynamic object
• Left Environment (env_l)
• Right Environment (env_r)

The message-statement also allows definition of message parameters of two forms:

• Message field
• Message field with value

The message name may be in identifier format (CallRequest) or string format ("Call Request"). The message field and value can be in identifier or string format.

Messages will be represented as arrows from source to destination. In Sequence Diagrams, message name is printed on the top of the arrow. Message parameters are printed below the arrow. The formatting of the message parameters is customizable (See the Tools->Options menu for details).

ADVANCED USES
• **Bidirectional message** interactions can be succinctly represented using the bidirectional form of the message statement. Bidirectional messages are represented with double headed arrows.

• **Important messages** may be represented using weighted arrows. These arrows are drawn with a thicker arrow.

• **Self messages** are specified by using the same entity as source and destination. “Self messages” are represented with arrows that return back to the originating axis.

• **Message chains** can be used to depict a sequence of message interactions.

• **Lost messages** are represented with an X next to a partial message arrow.

**EXAMPLES**

**Message 1**: `src_proc -> dest_proc`
Message 1 is being sent from `src_proc` to `dest_proc` component

```
"SYN" : client -> server
"SYN+ACK" : client <-> server
"ACK" : client -> server
```

The message sequence shown above demonstrates the use of the reverse arrow.

**Origination(subscriber_id, mode=NORMAL): v5_mg` -> dtmf_proc**
Origination message with parameters `subscriber_id` and `mode` is sent from `v5_mg` object to `dtmf_proc` component. Here the mode variable has been given a value of `NORMAL`.

**Origination(subscriber_id, mode=NORMAL): dtmf_proc <-> v5_mg**
This example uses a reverse arrow. The destination is on the left side of the reverse arrow. The source is on the right side of the reverse arrow.

**Setup(call_id, phone_num, priority=HIGH, latency=LOW): v5_mg -> env_r**

Here is an example of a message statement with parameters spread over multiple lines. The Setup message is being sent from the `v5_mg` to the right environment. The following parameters are being sent in the message:

- `call_id`
- `phone_num`
- `priority` with a value of `HIGH`
- `latency` with a value of `LOW`

"**Interview Employee**": "Human Resources" -> Department"
A string message name and source have been used in this example.

**HTTP_Get(url="http://www.EventHelix.com"): WebBrowser -> Internet**
A string parameter value is used in this example to represent the URL value.

"**Purchase Order**" ("Order Number" = "01-033-56"): User -> Server"
In this example, the message name, parameter type and value are all strings.
ADVANCED EXAMPLES

VoicePath: Caller <-> Called
A bidirectional voice path has been established between the caller and the called subscribers.

VoicePath: Caller <= Called
A bidirectional voice path has been established between the caller and the called subscribers. This time represented with a thick arrow.

"RTP Stream": Server -> Client
An RTP stream from the server to the client (represented with a weighted arrow).

"End of Dialing": CallHandler -> CallHandler
The call handler schedules an internal "End of Dialing" message.

[sip_style] "Invite" : SIP_Phone -> SIP_Proxy
A style, titled sip_style, is assigned to the "Invite" message. The message is formatted according to the style specification.

PathSetup: src -> dst <http://www.messagedef.com/PathSetup.html>
A message with a hyperlink specification is shown here. Clicking on the message name will open a browser window for the specified URL.

"RR CHANNEL REQUEST": Mobile-rr -> BSS-rr
A message shown here with source and destination tagging. The tags can be used to denote protocol stack layers exchanging the message. The representation of the tags is shown below:

```
chain
   "Call Setup": UT -> BTS
   "Call Setup": BTS -> BSC
   "Initial Call Message": BSC -> MSC
endchain
```
A chain statement enclosing messages is shown above. The chain messages are drawn in a single line.

TCPSegment : client -> X server
A TCP segment sent from the Client to the Server has been lost

CourierPackage : Courier => X Customer
A courier has lost a package (shown with a bold arrow).

TCPSegment (parameter1, parameter2): server X<- client
This example uses the reverse lost message arrow. It depicts that TCP Segment from client to server is lost.

CourierPackage(a_parameter, b_parameter): Customer X<= Courier
This example depicts bold reverse lost message arrow.

Chain statement
The Chain statement allows you to define a message chain to model a chain of messages that are triggered in a sequence. EventStudio attempts to draw all messages in a message in a single line.

The statement is useful in modeling scenarios where a sequence of messages is to be depicted and the actual handling takes place when the sequence of messages terminates. By using the chain statement, the sequence of messages is depicted on a single line in a sequence diagram. This reduces the clutter in the diagrams.

Multiple Message statements may be enclosed between the chain and endchain keywords:

**EXAMPLE**

```plaintext
chain
[rstyle] msg : a -> b
msg2(par1="one", par2): b -> c
[talk] conversation: c <-> d
endchain
```

The chain statement shown above will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines.

```plaintext
msg(par1="one", par2): a -> b -> c
```

The forward cascaded message statement shown above is a special case of chain statement where message opcode and parameters of chain elements is same and destination of one chain element is the source of next chain element. This type of chain statement will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines.

```plaintext
msg(par1="one", par2): a -> b -> c
```

The weighted forward cascaded message statement shown above is a special case of chain statement where message opcode and parameters of chain elements is same and destination of one chain element is the source of next chain element. This type of chain statement will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines.

```plaintext
msg(par1="one", par2): c <- b <- a
```

The reverse cascaded message statement shown above is a special case of chain statement where message opcode and parameters of chain elements is same and destination of one chain element is the source of next chain element. This type of chain statement will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines. The only difference from forward cascaded message statement is that reverse arrow is used in the syntax instead of a forward arrow.

```plaintext
msg(par1="one", par2): c <= b <= a
```

The weighted reverse cascaded message statement shown above is a special case of chain statement where message opcode and parameters of chain elements is same and destination of one chain element is the source of next chain element. This type of chain statement will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines. The only difference from forward cascaded message statement is that reverse arrow is used in the syntax instead of a forward arrow. The only difference from the previous example is that the message arrows are weighted.

```plaintext
msg(par1="one", par2): a <-> b <-> c
```
The cascaded bidirectional message statement shown above is a special case of chain statement where message opcode and parameters of chain elements is same and all chain elements are bidirectional message exchanges. This type of chain statement will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines. The difference from forward or reverse cascaded message statement is that double-headed arrows are used in the syntax and in the display.

```
msg(par1="one", par2): a <=> b <=> c
```

The weighted cascaded bidirectional message statement shown above is a special case of chain statement where message opcode and parameters of chain elements is same and all chain elements are bidirectional message exchanges. This type of chain statement will be drawn in a single line if the message arrows do not result in an overlap. If the message arrows result in an overlap, the message arrows are drawn in different lines. The difference from forward or reverse cascaded message statement is that double-headed arrows are used in the syntax and in the display. The only difference from the previous example is that the double-headed message arrows are weighted.

```
chain
 "Message 1": A-rtp -> B-rtp
 "Message 2": B-rtp -> C-rtcp
endchain
```

A message chain shown with tags.

**Multicast statement**

This statement is used to represent message sent from a given source to more than one destination at the same time. The source and destination can be any of the following:

- Component (i.e. component without any other objects inside it)
  - Eternal object
  - Dynamic object

The multicast-statement also allows definition of message parameters of two forms:

- Message field
- Message field with value

The message name may be in identifier format (CallRequest) or string format ("Call Request"). The message field and value can be in identifier or string format.

A multicast statement can be put only inside the feature-endfeature block in the FDL file.

Messages will be represented as arrows from source to destination. In Sequence Diagrams, message name is printed on the top of the arrow. Message parameters are printed below the arrow. The formatting of the message parameters is customizable (See the Tools->Options menu for details).

**EXAMPLES**
**src_proc multicasts Message1 to dest_proc1, dest_proc2**
Message1 is being sent from src_proc to dest_proc1 and dest_proc2 components

**v5_mgr multicasts Orig(subscriber_id, mode=NORMAL) to dtmf_proc, call_mgr**
Origination message with parameters subscriber_id and mode is sent from v5_mgr object to dtmf_proc component and call_mgr object. Here the mode variable has been given a value of NORMAL.

**WebTaxiServer multicasts "Taxi Request" to Taxi1, Taxi2**
A string message name has been used in this example.

**User multicasts "Purchase Order" ("01-033-56") to Server1, Server2**
In this example, the message name, parameter type and value are all strings.

**BaseStation multicasts "System Information" to UE1-bcch, UE2-bcch**
Tags are used to identify the BCCH channel in the multicast.

### Object Interactions

#### Invokes and Returns statements

The "Invokes" statement is used to model a method invocation on an object. The return from a method is modeled by the "returns" statement. You can specify parameters for the "invokes" as well as "returns" statements. The parameter specification syntax is similar to message parameters.

Invokes and returns statements may be nested.

EventStudio catches design errors like:

- Method returns are placed out of order. EventStudio makes sure that the methods return are made in a LIFO fashion (Last In First Out).
- Message receive is being attempted when a method execution is still active.

The method invocation is represented with an arrow similar to the message statement. Once a method is invoked, the axis is represented as a thin rectangular bar that lasts until method return.

The rectangular bar is not shown in the system interaction, subsystem interaction, module interaction and component interaction diagrams.

#### EXAMPLES

```plaintext
CallManager invokes Call.HandleInvite(msg="My Invite", bool send)
Call takes action "Verify the Invite Message"
Call invokes MessageHandler.Send(SIP_OK)
MessageHandler.Send returns (true)
Call.HandleInvite returns
```

A nested method invocation along with returns statements is shown above.
Call invokes Call.ProcessMessage(msg="My Invite", "bool send")
Call.ProcessMessage returns

An object invokes its own method.

Object Creation and Deletion

Create statement
This statement is used to create a dynamic object. The object that creates the dynamic object may or may not be specified. A dynamic object create statement can be put only inside the feature-endfeature block in the FDL file. Use a dynamic declaration statement to declare a dynamic object. Use the delete statement when a dynamic object needs to be deleted. A dynamic object created in a feature must be deleted before the feature ends.

The output corresponding to a create statement is shown as a dotted arrow from the creator entity to the created entity. Create is printed on the top of the arrow. And, the instance axis for the dynamic object starts just from the create arrow.

EXAMPLES

dynamic : call in frontend
feature "Call Setup"
  msg1 : phone -> call_mgr
  call_mgr creates call
  dialtone : call -> phone
  call_over : call -> call_mgr
  call_mgr deletes call
endfeature

Here, a dynamic object call is declared inside the frontend component. The Call Setup feature shows that the call_mgr object creates call object. Once the call is over, call_mgr deletes the call object.

dynamic : call in frontend
feature "Call Setup"
  msg1 : phone -> call_mgr
  call_mgr creates(call_id, phone_no) call
  dialtone : call -> phone
  call_over : call -> call_mgr
  call_mgr deletes call
endfeature

Here, create statement with parameters is shown. The output shows the parameter list below the create opcode in the sequence diagram, component interaction diagram, module interaction diagram, subsystem interaction diagram or system interaction diagram.

dynamic : HSS | MRFC in "IMS CN"
feature "IMS Call Setup"
  create HSS "I-CSCF", HSS take action "Identify S-CSCF"
  delete HSS
  create MRFC "S-CSCF", MRFC take action "Setup Conference"
  delete MRFC
Anonymous create and delete are used here to share a single axis between HSS and MRFC.

**Note:** When a light weight header is used, the create box for the dynamic object in the sequence diagram is drawn only when the dynamic object is created. If image based light weight header is selected, the dynamic object image is drawn only when the dynamic object is created. This applies to regular dynamic and reusable dynamic objects.

**Delete statement**

This statement is used to delete a dynamic object. The object that deletes the dynamic object may or may not be specified. The delete statement also allows a self-delete. A dynamic object delete statement can be put only inside the feature-endfeature block in the FDL file. Use a dynamic declaration statement to declare a dynamic object. Use the create statement when a dynamic object needs to be created. A dynamic object created in a feature must be deleted before the feature ends.

The output corresponding to a delete statement is shown as a big X mark. An arrow from the deleting object to the deleted object precedes the X mark if it is not the case of self-delete.

**EXAMPLES**

**call_over:** call -> call_mgr

**call_mgr deletes call**

Here, on receipt of the call_over message from call object, the call_mgr object deletes the call object.

**release_call:** rout_mgr -> call

**call deletes call**

Here, on receipt of the release_call message from the rout_mgr, the call object deletes itself.

**dynamic:** HSS | MRFC in "IMS CN"

**feature** "IMS Call Setup"

create HSS

"I-CSCF", HSS take action "Identify S-CSCF"

delete HSS

create MRFC

"S-CSCF", MRFC take action "Setup Conference"

delete MRFC

endfeature

Anonymous create and delete are used here to share a single axis between HSS and MRFC.

**Timers**

**Timer Start (One Shot and Periodic) statement**

This statement is used for starting a timer by an eternal or dynamic object. The timer name and the object starting the timer must be specified.

Use the timer stop statement for stopping a timer. Use timer timeout statement for showing the timeout for a timer. EventStudio issues a warning if it finds that a timer has been left running at feature end. For dynamic objects, EventStudio issues a warning if timers are left running at object deletion.
EventStudio supports one time and periodic timers. Only one timeout is permitted for "one time" timers. Periodic timers are allowed multiple timeouts.

The output of a timer start statement is shown as a starting point of a downward timer axis. The timer name is displayed on the top, at the timer start instant. The timer axis is shown active unless the timeout happens or the timer is stopped.

**EXAMPLES**

```fdl
feature "Call Setup"
  offhook : phone -> call_mgr
    call_mgr creates call
    call_start await_first_digit_timer
    digits : phone -> call
    call stops await_first_digit_timer

endfeature
```

Here, the call object starts the await_first_digit_timer (one time) timer to await first digit dialed by the subscriber.

```fdl
feature "Call Setup"
  offhook : phone -> call_mgr
    call_mgr creates call
    call_start periodic send_billing_pulse_timer
    timeout send_billing_pulse_timer
    . . .
    timeout send_billing_pulse_timer
    . . .
    timeout send_billing_pulse_timer

endfeature
```

Here, the call object starts the send_billing_pulse_timer periodic timer. Multiple timeouts are depicted for the timer.

**Timer Restart statement**

This statement is used for restarting an already existing timer started by any eternal or dynamic component. The timer name and the object stopping the timer must be specified. A timer restart statement can be put only inside the feature-endfeature block in the FDL file.

The output of a timer restart statement is shown as a continuation of a downward timer axis. The timer name is displayed on the top, at the timer restart instant. The timer axis is shown active unless the timeout happens or the timer is stopped.

**EXAMPLE**

```fdl
feature "Call Setup"
  offhook : phone -> call_mgr
    call_mgr creates call
    call_start await_digit_timer
    digits : phone -> call
    call restarts await_digit_timer
    timeout await_digit_timer
    . . .

endfeature
```

Here, the await_digit_timer has been restarted on receipt of digits from the phone.
**Timer Stop statement**

This statement is used for stopping an existing timer started by an eternal or dynamic object. The timer name and the object stopping the timer must be specified. A timer stop statement can be put only inside the feature-endfeature block in the FDL file. Use the timer start statement for starting a timer. Use timer timeout statement for showing the timeout for a timer. EventStudio issues a warning if it finds that a timer has been left running at feature end. For dynamic objects, EventStudio issues a warning if timers are left running at object deletion.

The output of a timer stop statement is shown as the end point of the downward timer axis. The timer name is displayed above a dotted reverse arrow at the timer stop instant.

**EXAMPLES**

```fdl
feature "Call Setup"
    offhook : phone -> call_mgr
    call_mgr creates call
    call starts await_first_digit_timer
digits : phone -> call
    call stops await_first_digit_timer
endfeature
```

Here, the call object stops the await_first_digit_timer timer on receipt of the first digit dialed by the subscriber.

**Timeout statement**

This statement is used to show the timeout of an existing timer started by an eternal or dynamic object. The timer name must be specified in the statement. A timeout statement can be put only inside the feature-endfeature block in the FDL file. Use the timer start statement for starting a timer. Use timer stop statement for stopping a timer. EventStudio issues a warning if it finds that a timer has been left running at feature end. For dynamic objects, EventStudio issues a warning if timers are left running at object deletion. A timer should either timeout or it should be explicitly stopped.

The output for a timer timeout statement is shown as the end point of the downward timer axis. The timer name is displayed above a forward arrow at the timer timeout instant.

**EXAMPLES**

```fdl
feature "Call Setup"
    offhook : phone -> call_mgr
    call_mgr creates call
    call starts await_first_digit_timer
    (* The calling subscriber does not dial any digit *)
timeout await_first_digit_timer
endfeature
```

Here, the calling subscriber does not dial any digit and the await_first_digit_timer timeout happens.

**Time Passage statement**

Passage of time can be depicted with three dots in the FDL. The display of the statement contains all the object axis, exiting timer axis and continuous action axis if present, to be dotted for a fixed length. The representation of time elapsed is shown in the following diagram.
A starts Timer
...
timeout Timer
Actions

Action statement
This statement is used to model a specific action taken by a single or a group of entities.

The statement takes two forms:

- **Single**: Action involving a single object, component, module, subsystem or system is specified.
- **Multiple**: Action involving multiple objects, components, modules, subsystems or systems is specified.

The output of an action statement is shown as a box with the action indicated inside it. System level action statement is included in system interaction diagram, subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Subsystem level action statement is included in subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Module level action statements are included in module interaction diagram, component interaction diagram and sequence diagram. Component level action statements are included in component interaction diagram and sequence diagram. The action statement font, color and line width may be changed by prefixing it with a style statement.

The "action" statement also supports hyperlinks in PDF based sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

**EXAMPLES**

call_mgr takes action "Obtain subscriber profile"
Here, the call_mgr object is shown to perform the action of obtaining the subscriber profile.

audit takes action "Check data consistency"
Here, the audit object is shown to perform the action of checking data consistency.

[act_style] client, server take action "Establish TCP Connection"
Here, the client and the server take a joint action "Establish TCP Connection". The statement also associates a style with the action statement. The style specifies the color, font, line width etc. for this action.

bank, broker, customer take action "Close the Loan"
The bank, the broker and the customer work together to close the loan.

a, b take action "SIP Call" <http://rfc.net/rfc3261.html>
Here the action box is also a hyper link pointing to the SIP RFC.

bsc1, bsc2 take action "Handover" <file://c:\Doc\handover_diagram.pdf>
The above statement links to a sequence diagram describing the handover in detail.

| system : Earth  |
| subsystem : "North America" in Earth, Asia in Earth |
| module : "United States" in "North America", Canada in "North America" |
| module : India in Asia |
| component : California in "United States", Nevada in "United States" |
| component : Quebec in Canada, Rajasthan in India |
| eternal : "Los Angeles" in California, "San Francisco" in California |
| eternal : "Las Vegas" in Nevada |
| eternal : Montreal in Quebec, Jaipur in Rajasthan |

feature "Bid for Events"
/* Component level action: 
- Represented by a box covering Los Angeles, San Francisco and Las Vegas axes.
- Included in component and lower level interaction diagram. */
California, Nevada take action "West Coast Bid for Olympics"

/* System level action: 
- Represented by a box covering all objects under Earth.
- Included in system level and lower interaction diagrams. */
Earth takes action "Earth Summit"

**Action Begin statement**

This statement is used to model an action started by a single entity or a group of entities.

The statement takes three forms:

- **Single**: Action involving a single object, component, module, subsystem or system is specified.
- **Single-Continuous**: Action involving a single object depicted by a continuous action box from “begin action” statement to “end action” statement.
- **Multiple**: Action involving multiple objects, components, modules, subsystems or systems is specified.

System level action begin statement is included in system interaction diagram, subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Subsystem level action begin statement is included in subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Module level action begin statements are included in module interaction diagram, component interaction diagram and sequence diagram. Component level action begin statements are included in component interaction diagram and sequence diagram.

The output of an "action begin" statement is shown as a box with begin as the title. The action is indicated inside the box. The action statement font, color and line width may be changed by prefixing it with a style statement. A continuous action is shown with an axis box spawning from “begin action” to “end action”.

The "action begin" statement also supports hyperlinks in PDF sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

**EXAMPLES**

```plaintext
call begins action "Feeding Ring back tone"
call ends action "Feeding Ring back tone"
```

Here, the call object is shown to initiate the action of feeding ring back tone. Later on, the call object ends feeding the ring back tone.

```plaintext
call begins continuous action "Feeding Ring back tone"
call ends continuous action "Feeding Ring back tone"
```

Here, the call object is shown to initiate the action of feeding ring back tone. Later on, the call object ends feeding the ring back tone. The main difference from the previous example is that a continuous action has been defined here. Continuous actions are represented with an action box that is displayed on the axis.
**Action End statement**

This statement is used to model an action completed by a single entity or a group of entities.

The statement takes three forms:

- **Single**: Action involving a single object, component, module, subsystem or system is specified.
- **Single-Continuous**: Action involving a single object depicted by a continuous action box from “begin action” statement to “end action” statement.
- **Multiple**: Action involving multiple objects, components, modules, subsystems or systems is specified.

System level action end statement is included in system interaction diagram, subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Subsystem level action end statement is included in subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Module level action end statements are included in module interaction diagram, component interaction diagram and sequence diagram. Component level action end statements are included in component interaction diagram and sequence diagram.

The output of an "action end" statement is shown as a box with begin as the title. The action is indicated inside the box. The action statement font, color and line width may be changed by prefixing it with a style statement. A continuous action is shown with an axis box spawning from “begin action” to “end action”.

The "action end" statement also supports hyperlinks in PDF sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

**EXAMPLES**

```plaintext
[my_style] client,server begin action "Security Association"
[my_style] client,server end action "Security Association"
```

The client and server objects jointly begin the action of setting up a "Security Association". Later on, the client and the server objects jointly end the "Security Association". The statement also associates a style with the action statement. The style specifies the color, font, line width etc. for this action.

```plaintext
Action End statement

This statement is used to model an action completed by a single entity or a group of entities.

The statement takes three forms:

- **Single**: Action involving a single object, component, module, subsystem or system is specified.
- **Single-Continuous**: Action involving a single object depicted by a continuous action box from “begin action” statement to “end action” statement.
- **Multiple**: Action involving multiple objects, components, modules, subsystems or systems is specified.

System level action end statement is included in system interaction diagram, subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Subsystem level action end statement is included in subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Module level action end statements are included in module interaction diagram, component interaction diagram and sequence diagram. Component level action end statements are included in component interaction diagram and sequence diagram.

The output of an "action end" statement is shown as a box with begin as the title. The action is indicated inside the box. The action statement font, color and line width may be changed by prefixing it with a style statement. A continuous action is shown with an axis box spawning from “begin action” to “end action”.

The "action end" statement also supports hyperlinks in PDF sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

**EXAMPLES**

```plaintext
call begins action "Feeding Ring back tone"

<http://c:\dialtone.pdf>
```

Here, the call object is shown to initiate the action of feeding ring back tone. Later on, the call object ends feeding the ring back tone. The "ends action" statement is hyperlinked to a file on the local drive.

```plaintext
call begins continuous action "Feeding Ring back tone"

<http://c:\dialtone.pdf>
```

Here, the call object is shown to initiate the action of feeding ring back tone. Later on, the call object ends feeding the ring back tone. The main difference from the previous example is that a continuous action has been defined here. Continuous actions are represented with an action box that displayed on the axis.

```plaintext
[my_style] client,server begin action "Security Association"
[my_style] client,server end action "Security Association"
```
The client and server objects jointly begin the action of setting up a "Security Association". Later on, the client and the server objects jointly end the "Security Association". The statement also associates a style with the action statement. The style specifies the color, font, line width etc. for this action.

**Resource Allocations**

**Resource Allocate statement**

This statement is used for allocating a resource by a single or a group of entities. The entities allocating the resource and the resource name must be specified. A resource allocate statement can be put only inside the feature-endfeature block in the FDL file. Use a resource free statement for deallocating a resource. EventStudio issues a warning when it finds that resources have not been deallocated at feature end. For dynamic objects, EventStudio issues a warning if the resources allocated by the object have not been freed before the object deletion.

The output of a resource allocate statement is shown as a box with "allocate" as the title and the resource name inside it.

**Note:** The strings used in this statement should not contain any leading or trailing blanks. Also words within the string should not be separated by more than one blank.

The statement takes two forms:

- **Single:** Resource allocation involving a single object, component, module, subsystem or system is specified.
- **Multiple:** Resource allocation involving multiple objects, components, modules, subsystems or systems is specified.

System level resource allocate statement is included in system interaction diagram, subsystem interaction diagram, module interaction diagram and sequence diagram. Subsystem level resource allocate statement is included in subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Module level resource allocate statements are included in module interaction diagram, component interaction diagram and sequence diagram. Component level resource allocate statements are included in component interaction diagram and sequence diagram.

**EXAMPLES**

`call_mgr allocates "Outgoing trunk"`

Here, the call manager object allocates an outgoing trunk resource for a call.

`call_mgr allocates "Outgoing trunk" <http://c:\referencedoc.pdf>`

The resource allocate statement also supports hyperlinks in PDF based sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

```
[my_style] client,server allocate "Encryption Coprocessor"
```

```
[my_style] client,server free "Encryption Coprocessor"
```

The client and server objects jointly allocate an "Encryption Coprocessor". Later on, the client and the server objects jointly free the "Encryption Coprocessor". The example also shows a style specification that can be
prefixed to the resource allocate and free statements. The "action end" statement also supports hyperlinks in PDF sequence diagrams. The hyperlinks may be used to link to files on the Internet or your local drive.

system : Earth
subsystem : "North America" in Earth, Asia in Earth
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan

feature "Bid for Events"

/* Component level resource allocate:
- Represented by a box with allocate as title and resource name inside it.
The box covers Los Angeles, San Francisco and Las Vegas axes.
- Included in component and lower level interaction diagram. */
California, Nevada allocate "Stadium for Olympics in West Coast"

California, Nevada free "Stadium for Olympics in West Coast"

/* System level resource allocate:
- Represented by a box with title allocate and resource name inside it
covering all objects under Earth.
- Included in system level and lower interaction diagrams. */
Earth allocates "Earth's Satellite"

Earth frees "Earth's Satellite"
endfeature

Resource Free statement

This statement is used for de-allocating a resource by a single or a group of entities. The resource must have been allocated earlier by the resource allocate statement. The entities deallocating the resource and the resource name must be specified in the statement. A resource free statement can be put only inside the feature-endfeature block in the FDL file. EventStudio issues a warning when it finds that resources have not been deallocated at feature end. For dynamic objects, EventStudio issues a warning if the resources allocated by the object have not been freed before the object deletion.

The output of a resource free statement is shown as a box with "free" as the title and the resource name inside it. Like resource allocate statement, resource free statement also supports user definable hyperlinks.

Note: The strings used in this statement should not contain any leading or trailing blanks. Also words within the string should not be separated by more than one blank.

The statement takes two forms:

- **Single**: Resource free involving a single object, component, module, subsystem or system is specified.
- **Multiple**: Resource free involving multiple objects, components, modules, subsystems or systems is specified.

System level resource free statement is included in system interaction diagram, subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Subsystem level resource
free statement is included in subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Module level resource free statements are included in module interaction diagram, component interaction diagram and sequence diagram. Component level resource free statements are included in component interaction diagram and sequence diagram.

EXAMPLES

```plaintext
[my_style] call_mgr frees "Outgoing trunk"
```

Here, the call manager object frees the outgoing trunk resource allocated earlier for a call. The example also shows a style specification that can be prefixed to the resource allocate and free statements.

```plaintext
client,server allocate "Encryption Coprocessor"
```

```plaintext
client,server free "Encryption Coprocessor"
```

The client and server objects jointly allocate an "Encryption Coprocessor". Later on, the client and the server objects jointly free the "Encryption Coprocessor".

```plaintext
system : Earth
subsystem : "North America" in Earth, Asia in Earth
module : "United States" in "North America", Canada in "North America"
module : India in Asia
component : California in "United States", Nevada in "United States"
component : Quebec in Canada, Rajasthan in India
eternal : "Los Angeles" in California, "San Francisco" in California
eternal : "Las Vegas" in Nevada
eternal : Montreal in Quebec, Jaipur in Rajasthan

feature "Bid for Events"
  California, Nevada allocate "Stadium for Olympics in West Coast"

  /* Component level resource free:
  - Represented by a box with free as title and resource name inside it. The
    box covers Los Angeles, San Francisco and Las Vegas axes.
  - Included in component and lower level interaction diagram. */
  California, Nevada free "Stadium for Olympics in West Coast"

  Earth allocates "Earth's Satellite"

  /* System level resource free:
  - Represented by a box with title free and resource name inside it covering
    all objects under Earth.
  - Included in system level and lower interaction diagrams. */
  Earth frees "Earth's Satellite"
endfeature
```

### State Transitions

#### State Change statement

This statement is used to represent state transitions of state machines. State for a single or a group of entities may be specified. The name of the entities and the name of the state to which transition is taking place, must be specified. A state change statement can be put only inside the feature-endfeature block in the FDL file. This statement is used during detailed design phase.
The output of a state change statement is shown as a hexagonal box with the name of state indicated inside it.

**Note:** The strings used in this statement should not contain any leading or trailing blanks. Also words within the string should not be separated by more than one blank.

The statement takes two forms:

- **Single:** State change involving a single object, component, module, subsystem or system is specified.
- **Multiple:** State change involving multiple objects, components, modules, subsystems or systems is specified.

System level state change statement is included in system interaction diagram, subsystem interaction diagram, module interaction diagram and sequence diagram. Subsystem level state change statement is included in subsystem interaction diagram, module interaction diagram, component interaction diagram and sequence diagram. Module level state change statements are included in module interaction diagram, component interaction diagram and sequence diagram. Component level state change statements are included in component interaction diagram and sequence diagram.

**EXAMPLES**

```
call state = "Collecting digits"
```
Here, the state of the call object state machine has been specified to change to collecting digits.

```
caller, called state = "Conversation"
```
The caller and called subscribers have jointly entered the "Conversation" state.

```
[my_style] "car dealership" state = "Awaiting Shipment"
```
Here a style prefix is used to control the layout of a state statement.

```
[my_style] customer, banker state = "Loan Closed"
```
Here a style prefix is used to control the layout of a joint state statement.

```
California, Nevada state = "Developed"
```
Here, the state of the California and Nevada state machines has been specified to change to developed. The output hexagonal box covers Los Angeles, San Francisco and Las Vegas axes. The output hexagonal box is included in component level and lower level interaction diagram.

**Sequence Groupings**

**Sequence Block Statement**

A sequence block is used to group multiple statements. A sequence block encloses multiple statements between the sequence-endsequence. Sequences are identified by their title. The title is also displayed in the PDF bookmarks. Sequence blocks may be nested.

A sequence block can be displayed in two modes, expanded or summarized. A sequence block is represented as a rectangle around the enclosed statements in the expanded mode. Nested sequences are represented with nested rectangles. In the summarized mode, the sequence block is represented as a rectangle with just the title; no interactions within the sequence block are shown.
The real value of the sequence statement is that it lets you control the level of detail in the document by choosing how to display a sequence. EventStudio supports the following options:

- Don't expand
- Expand first time only
- Always expand

A sequence block is automatically minimized if the sequence block has already been included in a previous scenario in the generated diagram. This feature helps minimize the size of a sequence diagram when many scenarios are included in a single diagram.

You may use this statement to define:

- A while loop that contains a sequence of statements that are repeated to accomplish a task.
- Additional level of detail that may be abstracted out in a high level diagram.

**EXAMPLES**

```plaintext
sequence "PDP Context Activation"
    "PDP Context Activation Request" : UT -> "Core Network"
    "Core Network" takes action "Establish Quality of Service"
    "PDP Context Activation Response" : UT <= "Core Network"
endsequence
```

The PDP context activation is modeled as a sequence. When expanded, the statements in this sequence block are rendered inside a rectangle that covers the entities participating in the sequence.

```plaintext
#define GET_DIGIT(_call, _dsp) sequence "Get Digit"
    "Request Digit" : _call -> _dsp;
    "Collected Digit" : _call <- _dsp;
    "More Digits" : _call -> _dsp;
    (* _call needs more digits. *)
endsequence
```

Sequence blocks and macros make a good fit. The GET_DIGIT macro contains the "Get Digit" sequence block.
Sequences can be nested as shown here. EventStudio automatically determines the bounding rectangle for each sequence.

**Remarks**

**Remark statement**

This statement allows the user to explain the feature flow. The remarks are enclosed within (* and *). FDL supports multi-line remark statement. It should be noted that a remark statement could be put only inside feature-endfeature block in the FDL file.

The remarks are displayed on the right side remark column of a sequence diagram. FDL associates remarks with the previous FDL statement.

**EXAMPLES**

```
routing_request(digits) : call_mgr -> rout_mgr
(* Call manager sends the routing request to routing manager *)
routing_response(destination_equipment_num) : rout_mgr -> call_mgr
```

Here, the remark corresponds to the message statement sending the routing request message containing dialed digits from call manager object to routing manager object.

**Block Remark statement**

This statement allows the user to explain the feature flow. FDL supports block remarks enclosed within [ and ]. FDL supports multi-line block remark statement. Block remarks are used to show milestones in the execution of a feature. They do not associate with any FDL statement. It should be noted that a block remark statement could be put only inside feature-endfeature block in the FDL file.

Block remarks are printed along the complete width of the PDF output page. Block remarks support the following formatting options:

- The text can be divided between paragraphs by simply leaving a blank line between the two paragraphs.
- Block remark text alignment can be changed from the Tools->Options menu. Supported alignment options are:
  - Left Align
  - Center Align
  - Right Align
  - Justify
- Verbatim (Line breaks are preserved)
EVENTS

event_response : rout_mgr -> call_mgr
ramping : call_mgr -> trm_call
[* Call routing complete, terminating subscriber being rung, answer awaited *]

Here, the block remark statement is not associated with any message statement. It indicates the milestone that the call is in ringing phase.

[This scenario describes call setup when the call is originated by a mobile subscriber. The document describes a detailed flow involving the base station, MSC, HLR and VLR.]

channel_setup : SubscriberTerminal -> BaseStation
AGCH : BaseStation -> SubscriberTerminal

This block remark statement has been used to provide an introduction to the scenarios. Note how the text has been divided into two paragraphs. The output scenario diagrams will preserve the division into paragraphs.

Comment statement

This statement is used to provide programmer documentation. FDL supports standard C-type comments enclosed within /* and */. FDL supports multi-line comment statement.

EventStudio ignores the standard C-type comments and no output is produced corresponding to the comment.

EXAMPLES

/* V5.2 originating call setup */
feature "Call Setup"
  msg1 : phone -> call_mgr
  msg2 : call_mgr -> core
endfeature

Here, the comment informs the programmer that the call setup feature for the v5.2 originating call is being covered.

Heading statement

A large sequence diagram may be subdivided into sections using the heading statement.

- The headings are displayed as a block remark.
- A bookmark to the headings is added in the left pane of a PDF sequence diagram file.
- Message numbering is reset when headings are encountered.

EXAMPLES

The syntax for headings is shown in the following example:

heading "Conversation Mode"

Issue statement

This statement is used to indicate unresolved issues that may exist. Another use may be to provide review comments. An issue statement can be put only inside the feature-endfeature block in the FDL file.
The output corresponding to an issue statement is shown in bold in the right side remarks column. Also, whenever a review document command is given, all the issue statements in the FDL file are reported in the output window.

**Note:** The strings used in this statement should not contain any leading or trailing blanks. Also words within the string should not be separated by more than one blank.

**EXAMPLES**

```issue "Call metering procedure to be defined"
```

Here, an issue statement has been used to indicate that the call metering procedure is yet to be finalized.

**Preprocessor**

FDL supports a powerful preprocessor that is similar to the C preprocessor. All preprocessor directives begin with a hash (#). The following preprocessor directives are supported:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &quot;architecture.fdl&quot;</code></td>
<td>Include an FDL file located in the current directory. Paths relative to the current directory may be used.</td>
</tr>
<tr>
<td><code>#include &lt;theme.fdl&gt;</code></td>
<td>Include an FDL file from the paths specified in Tools-&gt;Options-Preprocessor tab. Multiple include paths may be specified. They should be separated by a semicolon (;).</td>
</tr>
</tbody>
</table>
| `#define msg_ack(msg, a, b) "msg" : a -> b;\ (* msg sent from a to b *);\ "msg ack" : b -> a;\ (* b acknowledges msg *);` | - Macros may be defined with multiple parameters.  
- Multiple statements may be defined in a single line by separating them with a semicolon (;)  
- Macros may be split into multiple lines by using a semicolon-backslash-enter (;\) pattern.  
- Macro parameter substitution is allowed in macro text, strings, remarks and block remarks.  
- Macros may be nested. |
| `#define tone_feed(d, t, a) [* Feed d#t *];\ a takes action "Feeding d t";` | Token pasting operator hash (#) allows two macro parameters to be "pasted" together. |
| `#ifdef UMTS_RELEASE_7` | Conditional compilation is supported with the `#ifdef-#else-#endif` statement. |
| `#else` | |
| `#endif` | |
Symbols controlling the conditional compilation may be defined using a:

- `#define`, or
- Symbols defined in Tools->Options->Preprocessor. (Multiple symbols may be specified. The symbols must be separated by semicolons).

```
#ifndef UMTS_RELEASE_7
#endif
```

Conditional compilation is supported with the `#ifdef`--`#endif` statement. Symbols controlling the conditional compilation may be defined using a:

- `#define`, or
- Symbols defined in Tools->Options->Preprocessor. (Multiple symbols may be specified. The symbols must be separated by semicolons).

### Styles, Themes and Document Layout

#### Style declaration

Style specifications allow you to specify the formatting for message statements. The style declaration allows you to specify:

- The color as red, blue and green components.
- Font and font size for the message name text
- Font and font size for the parameter text
- Line pattern for the message arrow
- Line width for the message arrow

The ranges for individual parameters are specified below. Note that not all values need to be specified in a style declaration. Message defaults are used for unspecified values.

<table>
<thead>
<tr>
<th><strong>color</strong></th>
<th>Specifies the color to be used. The color is specified as a set of three floating point numbers between 0.0 and 1.0. The first one denotes the red component, the second specifies the green component and the last one specifies the blue component.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- Millions of colors can be defined in terms of the individual color component values. <strong>Colors</strong> appendix may be used to select from predefined colors.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>bgcolor</strong></th>
<th>Specifies the background color. The bgcolor attribute is supported in:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- Axis boxes in light weight header</td>
</tr>
<tr>
<td></td>
<td>- Begin/end action (single and joint)</td>
</tr>
<tr>
<td></td>
<td>- Allocate/free resource (single and joint)</td>
</tr>
</tbody>
</table>
- Take action (single and joint)
- Continuous action

<table>
<thead>
<tr>
<th><strong>textcolor</strong></th>
<th>Specifies the text color in all statements.</th>
<th><strong>Colors</strong> appendix may be used to select from predefined colors.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>paramcolor</strong></th>
<th>Specifies the parameter color in message and object interaction statements.</th>
<th><strong>Colors</strong> appendix may be used to select from predefined colors.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>font</strong></th>
<th>Specifies the font of the message name. The following font strings are supported.</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• &quot;Arial&quot;</td>
<td>• &quot;Arial Narrow&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>fontsize</strong></th>
<th>Specifies the size of the message name font in points.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>linewidth</strong></th>
<th>Specifies the width of the message arrow in points.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>linepattern</strong></th>
<th>Selects the line pattern for the message arrow. The options are:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• &quot;---&quot; (Normal line)</td>
<td>• &quot;- -&quot; (Dashed line)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>paramfont</strong></th>
<th>Specifies the font for message parameters. The font strings are same as the message name font specification.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>paramfontsize</strong></th>
<th>Specifies the size of the message parameter font in points.</th>
<th></th>
</tr>
</thead>
</table>
**Image**

Specifies a JPEG image associated with the style. The image specification is used when styles are applied to object, component and module declarations.

The image specified in the style is used in the column heading if "Lightweight header with column images" is selected from the "Tools menu->Options menu->Customize tab". The image selection in the header is based on the following rules:

<table>
<thead>
<tr>
<th>Eternal/Dynamic Object Images</th>
<th>Component Images</th>
<th>Module Images</th>
<th>Images Drawn in the Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>At least one eternal or dynamic object uses a style with image.</td>
<td>N/A</td>
<td>N/A</td>
<td>The eternal/dynamic objects that have associated images are drawn with the image.</td>
</tr>
<tr>
<td>No eternal or dynamic object has a style with an image.</td>
<td>At least one parent component has a style with an image.</td>
<td>N/A</td>
<td>Component image is used. The image is centered between all eternal and dynamic objects within the component.</td>
</tr>
<tr>
<td>No eternal or dynamic object has a style with an image.</td>
<td>No component has a style with an image.</td>
<td>At least one module has a style with an image.</td>
<td>Module level images are used. The images are centered between all entities contained in a module.</td>
</tr>
</tbody>
</table>

**match**

Specifies a regular expression that is used to automatically apply a style to statements that match the regular expression.

When a match attribute is present in a style, the style gets automatically gets applied to statements that match the specified regular expression.

For details about regular expressions, refer to the appendix on **Regular Expressions**

**EXAMPLE**

```
style sip: color=1.0,0.0,0.0, font=Times New Roman-Bold, fontsize='15",
    linewidth='4", linepattern= -.", paramfont="Courier New", paramfontsize="4"
style server: color="0.0,1.0,0.0", font="Times New Roman-Bold", image="server.jpg"
style mobile: color="0.0,0.0,1.0", font="Times New Roman-Bold", image="mobile.jpg"
```
In the above example, we have explicitly applied the style that we want to apply to a statement. EventStudio supports two ways to implicitly specify the styles:

- **Default Styles**
- **Regular Expression Style Filters**

The above example also illustrates the styling as applied to **Message Tagging**. EventStudio applies the udp_tag_style to the definition in the invite message.

**Default Styles**

Specifying a style with every statement can become tedious. EventStudio allows you specify the default style to be used when no style has been specified.

Simply define styles with the following names. EventStudio will automatically apply these styles to the appropriate statements and declarations. (Note that an explicit style specification will override the default style specification.)

<table>
<thead>
<tr>
<th>Default Style Name</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_invokes_style</td>
<td>Default style for invokes statements</td>
</tr>
<tr>
<td>default_returns_style</td>
<td>Default style of a return from an invokes-statement.</td>
</tr>
<tr>
<td>default_message_style</td>
<td>Default style for unidirectional and bidirectional message interactions.</td>
</tr>
<tr>
<td>default_cascaded_message_style</td>
<td>Default style for cascaded messages. If this style is specified, it applies to statements of the form:</td>
</tr>
<tr>
<td></td>
<td>• Msg : a -&gt; b -&gt; c</td>
</tr>
<tr>
<td></td>
<td>• Msg : a &lt;-&gt; b &lt;-&gt; c</td>
</tr>
<tr>
<td>default_tag_style</td>
<td>Default style for message tags.</td>
</tr>
<tr>
<td>default_multicast_style</td>
<td>Default style for multicast.</td>
</tr>
<tr>
<td>default_create_style</td>
<td>Default style for object creation statement.</td>
</tr>
<tr>
<td>default_delete_style</td>
<td>Default style for object deletion statement.</td>
</tr>
<tr>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>default_remark_style</td>
<td>Default style for remarks.</td>
</tr>
<tr>
<td>default_block_remark_style</td>
<td>Default style for block remarks.</td>
</tr>
<tr>
<td>default_timer_style</td>
<td>Default style for timer operations.</td>
</tr>
<tr>
<td>default_state_style</td>
<td>Default style for state transitions.</td>
</tr>
<tr>
<td>default_action_style</td>
<td>Default style for:</td>
</tr>
<tr>
<td></td>
<td>• Takes action</td>
</tr>
<tr>
<td></td>
<td>• Take action</td>
</tr>
<tr>
<td></td>
<td>• Begin action</td>
</tr>
<tr>
<td></td>
<td>• End action</td>
</tr>
<tr>
<td>default_continuous_action_style</td>
<td>Default style for:</td>
</tr>
<tr>
<td></td>
<td>• Begin continuous action</td>
</tr>
<tr>
<td></td>
<td>• End continuous action</td>
</tr>
<tr>
<td>default_allocate_style</td>
<td>Default style for resource allocation statement.</td>
</tr>
<tr>
<td>default_free_style</td>
<td>Default style for resource free statement.</td>
</tr>
<tr>
<td>default_system_style</td>
<td>Default axis style used in system level interactions diagrams.</td>
</tr>
<tr>
<td>default_subsystem_style</td>
<td>Default axis style used in subsystem level interactions diagrams.</td>
</tr>
<tr>
<td>default_module_style</td>
<td>Default axis style used in module level interactions diagrams.</td>
</tr>
<tr>
<td>default_component_style</td>
<td>Default axis style used in component level interactions diagrams.</td>
</tr>
<tr>
<td>default_eternal_style</td>
<td>Default axis style used for eternal objects.</td>
</tr>
<tr>
<td>default_dynamic_style</td>
<td>Default axis style used for dynamic objects.</td>
</tr>
</tbody>
</table>

**EXAMPLE**
Regular Expression Style Filters

You can use the match attribute in a style declaration to specify a regular expression. All statements that match the regular expression will automatically be assigned the style defining the match attribute. (For more information about regular expression matching, refer to the Regular Expressions appendix).

EXAMPLE

```plaintext
/* request_style is automatically applied to messages that end with the "Request". */
style request_style: color="1.0,0.0,0.0", font="Times New Roman-Bold",
   fontsize="15", linewidth="1", paramfont="Courier New",
   paramfontsize="4", match="Request$"

/* response_style is automatically applied to messages that end with the "Response". */
style response_style: color="0.0,1.0,0.0", font="Times New Roman-Bold",
   fontsize="15", linewidth="1", linepattern="- -", paramfont="Courier New",
   paramfontsize="4", match="Response$"

module: m
component : p in m
eternal : e in p, f in p

feature "Testing"
   /* The request_style is applied to the following statement because the message
      opcode matches the style regular expression (i.e. ends with Request). */
   ResourceAllocationRequest: e -> f
   /* The response_style is applied to the following statement because the message
      opcode matches the style regular expression. (i.e. ends with Response) */
   ResourceAllocationResponse : e <-> f
endfeature
```

Theme declaration

Themes provide an overall control over the document layout. Most Tools->Options commands can be overridden using themes.

- A theme declaration consists of a theme name followed by a sequence of attribute-value pairs. The attribute-value pairs might be specified across multiple lines. The syntax is:

  ```plaintext
  theme MyTheme: blockRemarkFormatting="right-align",
  paperSize="letter"
  theme LeftAlign: blockRemarkFormatting="left-align"
  ```

- Initial theme for a statement is specified as a modifier to the feature statement. The attributes specified in the initial theme override the "Tools->Options" settings. The syntax is:

  ```plaintext
  {MyTheme} feature "a feature"
  ```
**endfeature**

- Theme may be modified within the feature-endfeature block by placing the theme modifier on a line. The modified theme overrides the attributes that have been specified for the modified theme. For example:

```plaintext
{MyTheme} feature "a feature"
  [* Right aligned block remark *]
  [LeftAlign]
  [* Left aligned block remark *]
endfeature
```

- A theme statement based on the current settings can be automatically copied to the clipboard using the Tools->Copy Options as Theme command. Just issue a paste command to insert the theme statement at the desired location.

- The theme declaration supports the attribute-value pairs shown in the table below.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Possible values</th>
</tr>
</thead>
</table>
| messageParameterFormatting                          | "multiple-parameters-per-line-center-align"
|                                                    | "single-parameter-per-line-center-align"
|                                                    | "single-parameter-per-line-left-align"
| sequenceDiagramHeader                               | "multi-tier-header"
|                                                    | "single-tier-light-weight"
|                                                    | "single-tier-light-weight-with-images"
| displayParentInLightWeightHeader                    | true
|                                                    | false
| blockRemarkFormatting                               | left-align
|                                                    | center-align
|                                                    | "right-align"
|                                                    | justify
|                                                    | verbatim
| assignSequenceNumbersInSequenceDiagrams             | true
|                                                    | false
| includeRemarkColumnInSequenceDiagrams              | true
|                                                    | false
| paperSize                                           | letter
|                                                    | "legal"
|                                                    | "B"
|                                                    | "C"
|                                                    | "A4"
|                                                    | "A3"
|                                                    | "B4"
|                                                    | "<width-in-inch> inch <height-in-inch> inch"
|                                                    | or "<width-in-mm> mm <height-in-mm> mm" 1
| minimizeEMFImageWidth                               | true
|                                                    | false
| minimizeEMFImageLength                              | true
|                                                    | false
| sequenceDiagramLayout                               | auto-choose-between-portrait-landscape
|                                                    | landscape-only
|                                                    | portrait-only
| stateShape                                          | curved-rectangle
|                                                    | hexagon
| pageMarginPDF                                       | small
|                                                    | "medium"
|                                                    | "large"
| pageMarginEMF                                       | small
|                                                    | "medium"
|                                                    | "large"
| headingAndTitleFont                                 | Arial
|                                                    | "Arial Narrow"
|                                                    | "Courier New"
|                                                    | "Times New Roman"

1 e.g. `paperSize="5.5 inch 6.5 inch"` or `paperSize="60 mm 70 mm"`. 
### Page Break statement

This statement inserts a page break in PDF and EMF files. Use this statement whenever you wish to force the PDF or EMF output to start from a new page.

**Note:** This statement applies to only PDF and EMF files. It is ignored when generating HTML output.

**EXAMPLE**

```plaintext
pagebreak
[* Call Release Processing. *]
```

Here, a page break has been forced at the beginning of a new phase of a call.
Horizontal and Vertical Separator statements

You can insert a blank column by declaring a separator. An example is shown in the figure on the left. Space has been left between objects B and C.

A blank line can be inserted by just adding a separator between two statements. Refer to the example on the left.

---

eternal: A in p, B in p, separator in p, C in p, D in p
feature “F”
msg1: A -> B
msg2: A -> B
separator
msg3: A -> C
msg4: A -> D
HOW TO...

- How to Add a New Scenario
- How to Add a Document
- How to Add Review Comments to a Sequence Diagram
- How to Export to Microsoft Word
- How to Rename a Scenario
- How to Reorder Scenarios
- How to Delete a Scenario
- How to Create a Scenario Project from Existing FDL Files
- How to Create a New FDL File
- How to Generate Documents from Command Line
- How to Rename a Document
- How to Delete a Document

How to Add a New Scenario

1. Click on "Add Scenario" icon (Ctrl+R).
2. EventStudio will open Scenario Wizard: Scenario Name dialog box.
3. Enter the desired name for the Scenario and click the "Next" button.

   ![Scenario Wizard: Scenario Name](image)

4. EventStudio will display Scenario Wizard: Select FDL dialog box and will prompt you to select the FDL file from which the Scenario will be defined.
5. Click the "Browse" button. EventStudio will display Open dialog box.
6. Select the appropriate directory and then the FDL file to be used. Then, click the "Open" button.
7. EventStudio will display the Scenario Wizard: Select FDL dialog box. Click on "Next" button.
8. EventStudio will review the FDL file that was selected. If there are no errors, it will display all the legs of the first case statement found in the FDL file.
9. Choose the leg to be taken for the Scenario and click "Next" button. This will repeat for all legs in the FDL.
10. After the leg selections have been completed, EventStudio will display "Finish" button instead of "Next". Select the "Finish" button.
11. A Scenario using the FDL file has been added to the Scenario Project.

**How to Add a Document**

1. Click on the “Add Document” icon $(Ctrl+R)$.
2. EventStudio opens the Document Wizard and displays the selection of documents that can be added.
3. Choose the document type from:

<table>
<thead>
<tr>
<th>Document Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence Diagram</strong></td>
<td>Use case or message sequence chart in Adobe PDF or Microsoft Word Picture (EMF) format.</td>
</tr>
<tr>
<td></td>
<td>Interactions between entities are listed in time sequence.</td>
</tr>
<tr>
<td></td>
<td>All scenarios defined in the Scenario Project are included in the diagram.</td>
</tr>
<tr>
<td><strong>Interface Sequence Diagram</strong></td>
<td>Sequence diagram that just include interactions with a specified entity or entity type.</td>
</tr>
<tr>
<td></td>
<td>You can generate the following:</td>
</tr>
<tr>
<td></td>
<td>o System interface diagrams</td>
</tr>
<tr>
<td></td>
<td>o Subsystem interface diagrams</td>
</tr>
<tr>
<td></td>
<td>o Module interface diagrams</td>
</tr>
<tr>
<td></td>
<td>o Subsystem interface diagrams</td>
</tr>
<tr>
<td></td>
<td>o Eternal or dynamic object interaction diagrams</td>
</tr>
<tr>
<td><strong>Interaction Sequence Diagram</strong></td>
<td>Sequence diagrams that allow you to zoom in and out of different levels of abstraction. You can also generate documents that just contain message interactions for a specific tag (Refer to Message Tagging).</td>
</tr>
<tr>
<td></td>
<td>You can generate the following:</td>
</tr>
<tr>
<td></td>
<td>o Inter system interaction diagram</td>
</tr>
<tr>
<td></td>
<td>o Inter subsystem interaction diagram</td>
</tr>
<tr>
<td></td>
<td>o Inter module interaction diagram</td>
</tr>
<tr>
<td></td>
<td>o Inter component interaction diagram</td>
</tr>
<tr>
<td><strong>Message Filter Sequence Diagrams</strong></td>
<td>Sequence diagrams listing only messages that contain a specified string or match a regular expression.</td>
</tr>
</tbody>
</table>
|                                      | For example a regular expression of "\{Request\}|\{Response\}" will
generate a document with messages that contain the strings "Request" or "Response" in the message or method name.

| Unit Test Procedures | • Unit test procedures for a specified object.  
| | • The scenarios are tabulated as steps of actions and the associated expected results.  
| | • The procedures may be used for as a guide for code reviews.  

| Summary Statistics Document | • Object wise summary of all interactions.  
| | • The document summary can be generated for all objects or objects within a module or a component.  

| Statistics Document | • Matrix of module, component and object level statistics.  
| | • Total number of message interactions between two objects can be tracked by following the source object row and destination object column.  

| Collaboration Diagram | • Collaboration Diagram or Context Diagram in Adobe PDF or Microsoft Word Picture (EMF) formats.  
| | • The time component of the interactions is de-emphasized in these diagrams.  

| Interface Collaboration Diagram | • Analogous to Interface Sequence Diagrams  
| Interaction Collaboration Diagrams | • Analogous to Interaction Sequence Diagrams  

| Message Filter Collaboration Diagram XML Document | • Analogous to Message Filter Sequence Diagram  
| | • Export to XML  
| | • The generated XML can be transformed into code and other documents using XSLT.  

| Interface XML Document | • Analogous to Interface Sequence Diagrams  
| Interaction XML Document | • Analogous to Interaction Sequence Diagrams  

| Message Filter XML Document | • Analogous to Message Filter Sequence Diagram  

4. Select the document type and click “Next”. The Document Wizard then displays the Document options.
5. Change the document options as desired and click “Next” (In this example, we have changed the Graphics Format from Adobe PDF to Word Picture – EMF).

6. On the next screen click “Next”. The document gets added to the document list.
The newly added document. Right click and use the Generate Document menu.
How to Add Review Comments to a Sequence Diagram

- Install Adobe Reader X or greater.
- Check the Adobe Reader X option in Tools→Options→Customize tab.
- Generate the sequence diagrams or collaboration diagrams and e-mail the documents for review.
- The reviewers can use sticky notes and text highlighting to comment.

Click on comments to reveal the comment annotations

Sticky notes can be used for commenting and then responding to comments.

Text can be highlighted in several colors.
**How to Export to Microsoft Word**

1. Create a Microsoft Word Picture (EMF) as shown in *How to Add a Document*.
2. Open the Microsoft Word document where you wish to insert the document.
3. Position the cursor in the document where you want to insert the "Word Picture EMF" files.
4. Follow the instructions specified along with the screenshot:

   ![Image of Microsoft Word Picture menu]

   1. Click on the Picture menu in the Insert tab.
   2. Select all the documents using Control+A.
   3. Click **Insert and Link**.

   - With this option, Word inserts a copy of the document and also maintains a link to the original file. Changes to the file are automatically reflected in the document.
   - You can send a copy of the document to other users, without bundling the EMF files.

**Notes:**

1. The document in "Word Picture EMF" graphics format is generated as a directory of the name same as that of the document in the path where the scenario project is present.
2. The document consists of a set of files in this directory. Each page of the document is represented by a file. The file naming is: `<scenario number>_<page within scenario>_<scenario name>.emf`.

**Example**

Consider a Scenario Project "C:\MyDoc\ScenerioProject.scn" with "Docking" and "Undocking" scenarios. The document "Spaceport Collaboration Diagram Word" is generated in the directory "C:\MyDoc\Spaceport Collaboration Diagram Word\" where the files are sequentially defined as 0001_0001_Docking.emf, 0002_0001_Docking.emf, 0002_0001_Undocking.emf and so on.

**How to Rename a Scenario**

1. Select the scenario to rename and right click.
2. You have two choices:
   a. Choose the “Rename Scenario” option from the popup menu.
   b. Click the text part to reveal the edit box.

How to Reorder Scenarios

How to Delete a Scenario

How to Create a Scenario Project from Existing FDL Files

1. Click on "New and Recent…” icon \(\text{(Ctrl+N)}\).
2. EventStudio will open the dialog shown below. Select “Create New Scenario Project” button.
3. EventStudio invokes the Scenario Wizard. The rest of the procedure is same as How to Add a New Scenario.

How to Create a New FDL File

1. Click on the "New and Recent" icon (Ctrl+N)
2. EventStudio displays the New dialog box. Select the FDL File option.
3. Click the "Create New FDL" button.
4. EventStudio will open an editor window.
5. Start entering the FDL file’s contents.
6. When it is time to save, click the "Save" icon (Ctrl+S)
7. EventStudio displays the "Save As" dialog box.
8. Select the appropriate directory and enter the desired file name and click the "Save" button.
9. A new FDL file has been created and saved.

How to Generate Documents from Command Line

1. Add EventStudio into the default search path. See the help for your version of Windows for the exact command. On Windows 7, the steps are:
   a. From the start menu, select Computer
   b. From Computer click System properties.
   c. On the System Properties window, click the Advanced system settings.
   d. Select the Advanced tab from the displayed dialog.
   e. Click the Environment Variables button.
   f. Highlight the Path Variable section and click the Edit button.
   g. Add the EventStudio 6 path after a semicolon. The default path is:
      - “C:\Program Files (x86)\EventHelix.com\EventStudio System Designer 6” for Windows 7 64 bit.
      - “C:\Program Files\EventHelix.com\EventStudio System Designer 6” for Windows 7 32 bit.
2. Once the path has been set you can issue the following command to Generate All documents for a Scenario:
   a. Invoke the "Start->Run" command. Type "cmd" in the text box and click "OK" to invoke the command-line shell.

```
start/wait evstudio ScenarioName.scn /GenerateAll
```

3. You can add the command-line into a batch file to generate documents for multiple scenarios. For example:
```
start/wait evstudio Examples/XenonProject.scn /GenerateAll
start/wait evstudio Examples/SpacePortProject.scn /GenerateAll
start/wait evstudio Examples/WebTaxiProject.scn /GenerateAll
```

**Note:** The command-line mode can be invoked without the "start/wait" command. For example:
```
evstudio Examples/XenonProject.scn /GenerateAll
```

The only problem with this approach is that the command-line shell does not wait for the document generation to finish before returning to execute the next command. Needless to say, this approach is not suitable for scripting.

How to Rename a Document

1. Select the document to rename and right click.
2. You have two choices:
   a. Choose the “Rename Document” option from the popup menu.
   b. Click the text part to reveal the edit box.
How to Delete a Document

Right click and select option

Select and change text

Right click and select option
WHAT'S NEW IN EVENTSTUDIO SYSTEM DESIGNER 6

Message Endpoint Tagging

Specifying the Message Tags
The source and destinations of messages may be tagged. This is useful in:

- Identifying the source and destination ports
- The stack layer exchanging these messages
- Show the software module that will sending or receiving the message

A few examples of tags:

<table>
<thead>
<tr>
<th>Tag Style</th>
<th>Diagram</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSSMAP PAGING</td>
<td><img src="image.png" alt="Diagram" /></td>
<td>A-bssmap -&gt; B-bssmap</td>
</tr>
<tr>
<td>HTTPGet</td>
<td><img src="image.png" alt="Diagram" /></td>
<td>Client -&gt; Server - &quot;80&quot;</td>
</tr>
<tr>
<td>Cascade</td>
<td><img src="image.png" alt="Diagram" /></td>
<td>A-rr -&gt; B-cc -&gt; C-&quot;80&quot;</td>
</tr>
</tbody>
</table>

Note that tags may be specified as strings or identifiers.

Defining a Tag Style
Each type of tag can be individually formatted. The color, font and font size may be specified. The styles for tags should be named as <tag>_tag_style. A few examples are shown below:
Note that the tag style specification convention applies to identifiers as well as strings.

**Tag Export in XML**

A tag attribute has been added to source and destination tags in XML. A few examples are shown below:

```xml
<message>
  <id>1</id>
  <opcode>BSSMAP PAGING</opcode>
  <source tag="bssmap">A</source>
  <destination tag="bssmap">B</destination>
</message>

<multicast>
  <id>13</id>
  <opcode>SI</opcode>
  <source tag="bcch">A</source>
  <destination-list>
    <destination tag="bcch">B</destination>
    <destination tag="bcch">C</destination>
  </destination-list>
</multicast>

<message>
  <id>10</id>
  <opcode>HTTPGet</opcode>
  <source>Client</source>
  <destination tag="80">Server</destination>
</message>
```

**Generate tag specific diagrams**

Use the Interface documents to generate documents that only contain messages that use the tag. For example you could:

- Generate a document the only shows messages involving a particular protocol layer
- Limit the document contents to include interactions with a particular software entity
Specify the tag (prefixed with the dash) as the entity to be filtered. For example, the screenshot below defines a document that will only include messages with the "udp" tag.

Sample Enhanced to Demonstrate the Tag Feature
Sample #3 (Design a call flow) has been enhanced to demonstrate tags.

Sequence Enhancements

Simplified Sequence Definition
EventStudio 5 supported two types of sequence statements. They have now been merged into a single statement. EventStudio 6 automatically figures out the size of the sequence box based on the interactions in the sequence.

<table>
<thead>
<tr>
<th>EventStudio System Designer 5</th>
<th>EventStudio System Designer 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>eternal: a, b, c</td>
<td>eternal: a, b, c</td>
</tr>
<tr>
<td>feature &quot;Sequence Grouping&quot;</td>
<td>feature &quot;Sequence Grouping&quot;</td>
</tr>
<tr>
<td>sequence &quot;Setup&quot;</td>
<td>sequence &quot;Setup&quot;</td>
</tr>
<tr>
<td>IAM : a -&gt; b</td>
<td>IAM : a -&gt; b</td>
</tr>
<tr>
<td>b takes action &quot;Check digits&quot;</td>
<td>b takes action &quot;Check digits&quot;</td>
</tr>
<tr>
<td>&quot;new call&quot;: b -&gt; c</td>
<td>&quot;new call&quot;: b -&gt; c</td>
</tr>
<tr>
<td>ACM : a &lt;- b</td>
<td>ACM : a &lt;- b</td>
</tr>
<tr>
<td>endsequence</td>
<td>endsequence</td>
</tr>
<tr>
<td>a, b participate in sequence &quot;Release&quot;</td>
<td>a, b participate in sequence &quot;Release&quot;</td>
</tr>
<tr>
<td>REL : a -&gt; b</td>
<td>REL : a -&gt; b</td>
</tr>
<tr>
<td>RLC : a &lt;- b</td>
<td>RLC : a &lt;- b</td>
</tr>
<tr>
<td>endsequence</td>
<td>endsequence</td>
</tr>
<tr>
<td>endfeature</td>
<td>endfeature</td>
</tr>
</tbody>
</table>

No need to specify the participant entities.
Nest Sequences
EventStudio lets you nest sequences to clearly identify distinct phases in the sequence diagram. The sequences are also marked as nested bookmarks in the PDF sequence diagrams.

New Template to Demonstrate Sequence Enhancements
A new IMS-to-IMS call template (#20) demonstrates the use of the sequence enhancements in EventStudio 6.
**Feature Level Bookmarking in PDF**

The PDF bookmarks in sequence diagrams also include feature bookmarking. The book mark hierarchy now looks like:

<table>
<thead>
<tr>
<th>EventStudio System Designer 5</th>
<th>EventStudio System Designer 6 (refer to the PDF screenshot)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Diagram</strong></td>
<td><strong>Diagram</strong></td>
</tr>
<tr>
<td><strong>Scenarios</strong></td>
<td><strong>Features</strong></td>
</tr>
<tr>
<td><strong>Sequence Diagram</strong></td>
<td><strong>Sequence Diagram</strong></td>
</tr>
<tr>
<td>Success Scenario</td>
<td>Success Scenario</td>
</tr>
<tr>
<td>Failure Scenario</td>
<td>Failure Scenario</td>
</tr>
<tr>
<td>Another Scenario</td>
<td>Hello Goodbye</td>
</tr>
<tr>
<td></td>
<td>Another Scenario</td>
</tr>
</tbody>
</table>

---

![Sequence Diagram.pdf - Adobe Reader](image)

![Bookmarks](image)
**User Interface Improvements**

**Quick Access to Recent Scenario Projects at Startup**

The initial start dialog has been redesigned. You can now open recent Scenario Projects as well as create new Scenario Projects from the new dialog.

---

**Separate Menus for Recent Scenario Projects and Recent FDL Files**

---
Choose Between Curved and Hexagonal State Representation

EventStudio 6 now rendered states as curved rectangles. An option has been added to choose between curved rectangles and hexagon representation of states.

Other Enhancements

Increased Flexibility in Method Invoke and Return

Method invocation rules have been relaxed to make them compatible with sequence diagram generation from trace messages (http://eventhelix.github.com/trace-to-sequence-diagram/). Method calls to caller classes are permitted.

Configuration Management Friendly Scenario Project Saving

Scenario Project saving has now been optimized for configuration management. The Scenario Projects are updated only when scenario are modified.

HTML5 Compliant Unit Test Procedure and Summary Documents

All HTML documents are now HTML5 compliant. This includes:

- Unit test procedures
- Summary documents
- Statistics documents
- Document Index
FEATURES ADDED IN EVENTSTUDIO SYSTEM DESIGNER 5

XML Export

Full XML Export
Generate XML output from the scenarios. The generate XML can be used to:

- Generate custom diagrams and documents using XSLT

Hierarchical Decomposition Filtered XML Export
Generate XML diagrams at different levels of detail:

- System level XML
- Subsystem level XML
- Module level XML
- Component level XML
- Object level XML

Interface Filtered XML Export
Generate XML files that are restricted to interactions involving a:

- System or system type
- Subsystem or subsystem type
- Module or module type
- Component or component type
- Object or object type

Regular Expression Filtered XML Export
XML output can be generated for messages and object interactions that match a specified regular expression.

System Hierarchical Decomposition Enhancements

New Levels of Hierarchical Decomposition: System and Subsystem
EventStudio 4.0 supports three levels of decomposition:

- Module
- Processor/Component
- Eternal or Dynamic Object

EventStudio now supports five levels of decomposition:

- System
- Subsystem
- Module
- Component
- Eternal or Dynamic Object

**System Level and Subsystem Level Interaction Diagrams**

EventStudio adds two new diagram types:

- System level interaction diagrams show only system level interactions
- Subsystem level interaction diagrams show interactions between subsystems

**Choose from One to Five Levels of System Decomposition**

EventStudio 4.0 required a three level decomposition of the system (module, component and object).

EventStudio now allows you to choose from 1 to 5 levels of decomposition. You may just specify a single level model (eternal and dynamic objects only).

**Model Actions and Resource Allocations at All Five Levels of Decomposition**

Entities at all the five levels, systems, subsystems, modules, components and eternal or dynamic objects can perform actions, resource allocations and state transitions.
- System level actions, resource allocations and state transitions are included in system, subsystem, module, component and object interaction diagram.

- Subsystem level actions, resource allocations and state transitions are included in subsystem, module, component and object interaction diagram.

- Module level actions, resource allocations and state transitions are included in module, component and object interaction diagram.

- Component level actions, resource allocations and state transitions are included in component and object interaction diagram.

**Sequence Diagram Style Enhancements**

**Control the Look and Feel with Default Styles**

Look and feel of the display of every statement can be influenced by specifying style for the statement. Attributes like message display color, message name color, message parameter color, message name font, message name font size, parameter font, parameter font size, message display line pattern, message display line width can be defined for a particular message statement by defining a style and then applying it to the message statement. A style can be defined by using style declaration statement. Specifying style for every statement may be cumbersome in a sequence diagram. EventStudio supports default style specification for different types of statements.

For example, you can define a default style for all action statements. This results in EventStudio applying the same style for all action statements if no style is associated with the statement. Refer to section on styles, themes and document layout for details on how to define default style for various statements.

**Automatically Apply Styles by Regular Expression Matching Content**

The regular expression match attribute in the style declaration can be used to control the look and feel of a statement. EventStudio now supports applying a particular style to a statement if the regular expression defined in the definition of style matches the main text content in the statement. Refer to section on styles, themes and document layout for details on how to define and apply styles with regular expressions.

**Specify Color and Fonts for Block Remarks and Remarks**

Block remark statement now supports style specification.

- The block remark box color can be specified by the color attribute of style associated with the statement.
- The block remark background color can be specified by background color attribute of style.
- The block remark content color can be specified by the text color attribute of style.
- The block remark text font can be specified by the font attribute of style.

**More Color and Line Pattern Options in Styles**

- **Background Color:** The background color in axis boxes, block remark statement, action statement, action begin and action end, continuous action statement and resource allocate/release statements can now be specified by using background color option of style.
• **Text Color:** The text color of text of message, create, delete, resource allocate, resource free, action, continuous action, block remark, remark, timer, invokes and return statements can now be specified by using text color option of style.

• **Parameter Color:** The parameter color of message, create and invokes statement parameters can now be specified by using parameter color option of style.

• **Dotted Line Pattern:** The line pattern of the message arrow can now be specified to be dotted in the line pattern option of style.

An example sequence diagram depicting various style options for different statements is shown in the figure given below.

**Diagram Layout Optimization**

**Automatically Remove Unused Instance Axes**

Interface diagrams and large diagrams might contain instance axes that are not used in the diagram. When unused instance axes removal is enabled, these instances are automatically removed from the diagram.
Minimize Diagram Width and Length for Microsoft Word

EventStudio supports an option to limit the width of EMF images generated by it based on the total number of columns and selected remark width. When this option is selected, EventStudio does not auto expand the column width to fill the specified paper size. This feature helps in generating compact EMF images that can be inserted in Microsoft Word documents.

EventStudio supports an option to limit the length of EMF images generated by it based on the length of the content on the page. When this option is selected, EventStudio minimizes the length of the page to the end of the content. This feature helps in generating compact EMF images that can be inserted in Microsoft Word documents.
Group Interactions with the Sequence Statement

Complete sequences are shown the first time they occur in a sequence diagram. Subsequent occurrences of the sequence are shown as a simple clickable box. Clicking on the box takes the user to the expanded sequence.

---

module: Module_01
component: Component_01 in Module_01
eternal: a in Component_01, b in Component_01, c in Component_01
feature "Sequence Grouping"
  sequence "Call Setup"
    IAM : a -> b
    b takes action "Check digits"
    ACM : a <- b
  endsequence
  case
    leg "Called Party Free":
      a, b take action "Set up the voice path"
    leg "Called Party Busy":
      a, b take action "Feed busy tone"
  endcase
  sequence "Call Release"
    REL : a -> b
    RLC : a <- b
  endsequence
endfeature

---

Group Interactions in to Sequences
EventStudio supports sequence-endsequence block statement to group multiple statements. The sequence blocks are represented as shown in the figure above.

Auto Minimize Sequence When Repeated in Scenarios
EventStudio supports an option for minimizing sequence blocks every time a sequence block statement is encountered or sequence block is encountered after its first occurrence. The minimized version of the sequence block is a box with sequence name inside it in the display as shown in the figure given below.

Click on PDF to View Minimized Sequence
EventStudio supports link from minimized sequence block to sequence block display on its first time occurrence by clicking on the minimized sequence block display box in PDF sequence diagrams.

Message Enhancements

Model Message Loss
Lost messages can be clearly depicted with this statement.

![Lost Message: A -> X B](image)

Message Cascade: A -> X B

Model Message Cascades
Model message sequences where the same message is forwarded across multiple objects. This message cascade is modeled with a single message statement.

![Message Cascade: A -> B -> C](image)

Message Cascade: A -> B -> C

Object Interaction Enhancements

Hierarchically Classify Object Interactions
EventStudio supports dynamic object create and delete across components. It also supports “invokes” and “returns” statements across components. Since it supports hierarchical decomposition at the level of component, module, subsystem and system, inter-component, inter-module, inter-subsystem and inter system interaction sequence diagrams shall be enhanced to display create, delete, invokes and returns statement.

Generate Object Interaction Diagrams at Different Levels of Abstraction
EventStudio can generate sequence diagrams and collaboration diagrams at 5 different levels:

1. Full detail
2. Component level
3. Module level
4. Subsystem level
5. System level
Specify Object Creation Parameters
Now you can specify the parameters with the create statement. Now you can model object creation, right down to the constructor parameters.

Large Scale System Modeling

Share a Single Instance Axis between Multiple Objects
EventStudio allows many dynamic objects to share the same single instance axis slot. This can be achieved by separating the dynamic objects by the pipe (|) symbol in dynamic object declaration statement. The scenario should make sure that only one of these multiple dynamic objects is active at any time. A dynamic object axis begins at object create and ends at object delete. Dynamic object create is displayed by create message from creator object to the dynamic object box if light weight header is selected. If image based header is selected, the dynamic object image is drawn instead of a box. Dynamic object delete is depicted by delete message from source object followed by a cross at axis end. Anonymous dynamic object create and delete are also supported by EventStudio. The display of multiple objects sharing a single instance axis is as shown below.

Anonymous Object Create and Delete
Dynamic axis begin can be achieved by anonymous object create. The display is similar to object created by a creator object with the difference that the create message arrow is not drawn. Dynamic axis end can be achieved by anonymous object delete. The display is similar to the object delete by a source object with the difference that the delete message arrow is not drawn.

Model Continuous Actions
Model actions that have a distinct beginning and ending can be depicted with the new continuous action support. The actions are represented as shown in the sequence diagram on the left.

Command Line Enhancements

Specify Conditional Defines at Command Line
Script the conditional compilation defines via command-line, thus generating different versions of the document automatically.

Define Include Paths from Command Line
Change the header files by just changing the include path. Generate different versions of the document by just selecting a new set of header files from the command line.
**Other Enhancements**

**Object wise Summary Enhancements**
The layout of object wise summary document has been modified in EventStudio. The document covers all scenarios in order. Under each scenario, action remark table is depicted for each object.

**Unit Test Procedure Enhancements**
The layout of unit test procedure document has been modified in EventStudio. The document covers all scenarios with legs taken is depicted in order. Under this, unit tests are covered in the form of action result table.

**Hyperlink Support Extended to all Action Statements**
All action statements i.e. action, action begin, action end, resource allocate, resource free and state statements now support user definable hyperlinks.

**Simplified EMF File Update**
The EMF file naming has been modified to ease the update of EMF files after small additions.

**Component Box in Light Weight Header**
EventStudio supports displaying a box around all the objects of a component when component box option is selected for light weight header. An example is shown in the figure given below.

**Model Passage of Time**
Passage of time can be depicted with three dots in the FDL. The display of the statement contains all the object axis, exiting timer axis and continuous action axis if present, to be dotted for a fixed length. The representation of time elapsed is shown on the right.

**Control Spacing with Separators**
You can insert a blank column by declaring a separator. An example is shown in the figure on the left. Space has been left between objects B and C.

A blank line can be inserted by just adding a separator between two statements. Refer to the example on the left.

**Copy possible theme options values as comment**

A theme statement based on the current settings can be automatically copied to the clipboard using the Tools → Copy Options as Theme command. EventStudio supports copying all possible theme options values as comment to the clipboard followed by current theme options settings for layout and fonts.
## REGULAR EXPRESSIONS

### Regular Expression Examples

<table>
<thead>
<tr>
<th>Regular Expression Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docking</td>
<td>Matches all messages containing &quot;Docking&quot; anywhere in the message, e.g. ReadyForDocking, DockingCommand</td>
</tr>
<tr>
<td>^Docking</td>
<td>Matches all messages beginning with &quot;Docking&quot;, e.g. DockingCommand, DockingRejected</td>
</tr>
<tr>
<td>\undocking</td>
<td>Matches all messages containing &quot;undocking&quot;. Case is ignored in the comparison, e.g. UndockingAllowed, tawait_undocking_finished</td>
</tr>
<tr>
<td>Response$</td>
<td>Matches all messages ending with &quot;Response&quot;, e.g. OrbitDeallocResponse, EndDockingResponse</td>
</tr>
<tr>
<td>{Request}</td>
<td>{Response}$</td>
</tr>
<tr>
<td>^:.r</td>
<td>Matches all messages with &quot;r&quot; as the second character, e.g. OrbitAllocRequest, ProceedForUndocking</td>
</tr>
<tr>
<td>^[DU]</td>
<td>Matches messages beginning with D or U. e.g. DockingRequest, UndockingRequest</td>
</tr>
<tr>
<td>^[^DU]</td>
<td>Matches messages not beginning with D or U. e.g. ReadyForDocking, OrbitRequest</td>
</tr>
<tr>
<td>&lt;Act</td>
<td>Matches message containing words beginning with Act. e.g. ActivateSession, &quot;Demand Action&quot;</td>
</tr>
<tr>
<td>ion&gt;</td>
<td>Matches messages containing words ending with ion. e.g. &quot;Demand Action&quot;, &quot;Spread ion message&quot;</td>
</tr>
<tr>
<td>^:[0-9]</td>
<td>Matches messages that have a digit in the second character. e.g. M1Request, &quot;Q2 Report&quot;</td>
</tr>
<tr>
<td>\s</td>
<td>Matches message that contain white spaces. e.g. &quot;Demand Action&quot;, &quot;Invoke Call Request&quot;</td>
</tr>
<tr>
<td>^{Setup}{Release}([a-z]#[([0-9]#]$</td>
<td>Matches messages that begin with &quot;Setup&quot; or &quot;Release&quot; and contains lower case word followed by a number. e.g. Setup_call_53, Release_message_23,</td>
</tr>
<tr>
<td>Pattern</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td>^</td>
<td>Match the beginning of line</td>
</tr>
<tr>
<td>$</td>
<td>Match the end of line</td>
</tr>
<tr>
<td>.</td>
<td>Match any character</td>
</tr>
<tr>
<td>[ ]</td>
<td>Match characters in set</td>
</tr>
<tr>
<td>[^ ]</td>
<td>Match characters not in set</td>
</tr>
<tr>
<td>?</td>
<td>Match previous pattern 0 or 1 times (greedy)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>@</td>
<td>Match previous pattern 0 or more times (non-greedy)</td>
</tr>
<tr>
<td>#</td>
<td>Match previous pattern 1 or more times (non-greedy)</td>
</tr>
<tr>
<td>*</td>
<td>Match previous pattern 0 or more times (greedy)</td>
</tr>
<tr>
<td>+</td>
<td>Match previous pattern 1 or more times (greedy)</td>
</tr>
<tr>
<td>{}</td>
<td>Group characters to form one pattern</td>
</tr>
<tr>
<td>()</td>
<td>Group and remember</td>
</tr>
<tr>
<td>\</td>
<td>Quote next character (only of not a-z)</td>
</tr>
<tr>
<td>&lt;</td>
<td>Match beginning of a word</td>
</tr>
<tr>
<td>&gt;</td>
<td>Match end of a word</td>
</tr>
<tr>
<td>\x##</td>
<td>Match character with ASCII code ## (hex)</td>
</tr>
<tr>
<td>\N###</td>
<td>Match ASCII code #### (dec)</td>
</tr>
</tbody>
</table>

Matches messages starting with `Error` and followed by hex numbers. e.g. Error_0xAA_0x01_0x555, Error_0xAB
<table>
<thead>
<tr>
<th>\o###</th>
<th>Match ASCII code</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>Match \a</td>
</tr>
<tr>
<td>\r</td>
<td>Match 0x13 (cr)</td>
</tr>
<tr>
<td>\b</td>
<td>Match \b</td>
</tr>
<tr>
<td>\t</td>
<td>Match 0x09 (tab)</td>
</tr>
<tr>
<td>\f</td>
<td>Match \f</td>
</tr>
<tr>
<td>\v</td>
<td>Match \v</td>
</tr>
<tr>
<td>\n</td>
<td>Match 0x10 (lf)</td>
</tr>
<tr>
<td>\e</td>
<td>Match escape (^E)</td>
</tr>
<tr>
<td>\s</td>
<td>Match whitespace (cr/lf/tab/space)</td>
</tr>
<tr>
<td>\S</td>
<td>Match nonwhitespace (!\S)</td>
</tr>
<tr>
<td>\w</td>
<td>Match word character</td>
</tr>
<tr>
<td>\W</td>
<td>Match non-word character</td>
</tr>
<tr>
<td>\d</td>
<td>Match digit character</td>
</tr>
<tr>
<td>\D</td>
<td>Match non-digit character</td>
</tr>
<tr>
<td>\U</td>
<td>Match uppercase</td>
</tr>
<tr>
<td>\L</td>
<td>Match lowercase</td>
</tr>
<tr>
<td>\C</td>
<td>Match case sensitively from here on</td>
</tr>
<tr>
<td>\c</td>
<td>Match case ignore from here on</td>
</tr>
</tbody>
</table>
## COLORS

<table>
<thead>
<tr>
<th>Color</th>
<th>RGB Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLACK</td>
<td>&quot;0.0,0.0,0.0&quot;</td>
</tr>
<tr>
<td>DIMGRAY</td>
<td>&quot;0.41,0.41,0.41&quot;</td>
</tr>
<tr>
<td>DIMGREY</td>
<td>&quot;0.41,0.41,0.41&quot;</td>
</tr>
<tr>
<td>GRAY</td>
<td>&quot;0.50,0.50,0.50&quot;</td>
</tr>
<tr>
<td>GREY</td>
<td>&quot;0.50,0.50,0.50&quot;</td>
</tr>
<tr>
<td>DARKGREY</td>
<td>&quot;0.66,0.66,0.66&quot;</td>
</tr>
<tr>
<td>DARKGRAY</td>
<td>&quot;0.66,0.66,0.66&quot;</td>
</tr>
<tr>
<td>SILVER</td>
<td>&quot;0.75,0.75,0.75&quot;</td>
</tr>
<tr>
<td>LIGHTGRAY</td>
<td>&quot;0.83,0.83,0.83&quot;</td>
</tr>
<tr>
<td>LIGHTGREY</td>
<td>&quot;0.83,0.83,0.83&quot;</td>
</tr>
<tr>
<td>GAINSBORO</td>
<td>&quot;0.86,0.86,0.86&quot;</td>
</tr>
<tr>
<td>WHITESMOKE</td>
<td>&quot;0.96,0.96,0.96&quot;</td>
</tr>
<tr>
<td>WHITE</td>
<td>&quot;1.00,1.00,1.00&quot;</td>
</tr>
<tr>
<td>ROSYBROWN</td>
<td>&quot;0.74,0.56,0.56&quot;</td>
</tr>
<tr>
<td>INDIANRED</td>
<td>&quot;0.80,0.36,0.36&quot;</td>
</tr>
<tr>
<td>BROWN</td>
<td>&quot;0.65,0.16,0.16&quot;</td>
</tr>
<tr>
<td>FIREBRICK</td>
<td>&quot;0.70,0.13,0.13&quot;</td>
</tr>
<tr>
<td>LIGHTCORAL</td>
<td>&quot;0.94,0.50,0.50&quot;</td>
</tr>
<tr>
<td>MAROON</td>
<td>&quot;0.50,0.0,0.0&quot;</td>
</tr>
<tr>
<td>DARKRED</td>
<td>&quot;0.55,0.0,0.0&quot;</td>
</tr>
<tr>
<td>RED</td>
<td>&quot;1.00,0.0,0.0&quot;</td>
</tr>
<tr>
<td>SNOW</td>
<td>&quot;1.00,0.98,0.98&quot;</td>
</tr>
<tr>
<td>SALMON</td>
<td>&quot;0.98,0.50,0.45&quot;</td>
</tr>
<tr>
<td>MISTYROSE</td>
<td>&quot;1.00,0.89,0.88&quot;</td>
</tr>
<tr>
<td>TOMATO</td>
<td>&quot;1.00,0.39,0.28&quot;</td>
</tr>
<tr>
<td>DARKSALMON</td>
<td>&quot;0.91,0.59,0.48&quot;</td>
</tr>
<tr>
<td>ORANGERED</td>
<td>&quot;1.00,0.27,0.0&quot;</td>
</tr>
<tr>
<td>CORAL</td>
<td>&quot;1.00,0.50,0.31&quot;</td>
</tr>
<tr>
<td>LIGHTSALMON</td>
<td>&quot;1.00,0.63,0.48&quot;</td>
</tr>
<tr>
<td>SIENNA</td>
<td>&quot;0.63,0.32,0.18&quot;</td>
</tr>
<tr>
<td>CHOCOLATE</td>
<td>&quot;0.82,0.41,0.12&quot;</td>
</tr>
<tr>
<td>SADDLEBROWN</td>
<td>&quot;0.55,0.27,0.7&quot;</td>
</tr>
<tr>
<td>Color</td>
<td>RGB Values</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SEASHELL</td>
<td>&quot;1.00,0.96,0.93&quot;</td>
</tr>
<tr>
<td>SANDYBROWN</td>
<td>&quot;0.96,0.64,0.38&quot;</td>
</tr>
<tr>
<td>PEACHPUFF</td>
<td>&quot;1.00,0.85,0.73&quot;</td>
</tr>
<tr>
<td>PERU</td>
<td>&quot;0.80,0.52,0.25&quot;</td>
</tr>
<tr>
<td>LINEN</td>
<td>&quot;0.98,0.94,0.90&quot;</td>
</tr>
<tr>
<td>DARKORANGE</td>
<td>&quot;1.00,0.55,0.0&quot;</td>
</tr>
<tr>
<td>BISQUE</td>
<td>&quot;1.00,0.89,0.77&quot;</td>
</tr>
<tr>
<td>TAN</td>
<td>&quot;0.82,0.71,0.55&quot;</td>
</tr>
<tr>
<td>BURLYWOOD</td>
<td>&quot;0.87,0.72,0.53&quot;</td>
</tr>
<tr>
<td>ANTIQUEWHITE</td>
<td>&quot;0.98,0.92,0.84&quot;</td>
</tr>
<tr>
<td>NAVAJOWHITE</td>
<td>&quot;1.00,0.87,0.68&quot;</td>
</tr>
<tr>
<td>BLANCHEDALMOND</td>
<td>&quot;1.00,0.92,0.80&quot;</td>
</tr>
<tr>
<td>PAPAYAWHIP</td>
<td>&quot;1.00,0.94,0.84&quot;</td>
</tr>
<tr>
<td>MOCCASIN</td>
<td>&quot;1.00,0.89,0.71&quot;</td>
</tr>
<tr>
<td>WHEAT</td>
<td>&quot;0.96,0.87,0.70&quot;</td>
</tr>
<tr>
<td>OLDLACE</td>
<td>&quot;0.99,0.96,0.90&quot;</td>
</tr>
<tr>
<td>ORANGE</td>
<td>&quot;1.00,0.65,0.0&quot;</td>
</tr>
<tr>
<td>FLORALWHITE</td>
<td>&quot;1.00,0.98,0.94&quot;</td>
</tr>
<tr>
<td>GOLDENROD</td>
<td>&quot;0.85,0.65,0.13&quot;</td>
</tr>
<tr>
<td>DARKGOLDENROD</td>
<td>&quot;0.72,0.53,0.4&quot;</td>
</tr>
<tr>
<td>CORNSILK</td>
<td>&quot;1.00,0.97,0.86&quot;</td>
</tr>
<tr>
<td>GOLD</td>
<td>&quot;1.00,0.84,0.0&quot;</td>
</tr>
<tr>
<td>KHAKI</td>
<td>&quot;0.94,0.90,0.55&quot;</td>
</tr>
<tr>
<td>LEMONCHIFFON</td>
<td>&quot;1.00,0.98,0.80&quot;</td>
</tr>
<tr>
<td>PALEGOLDENROD</td>
<td>&quot;0.93,0.91,0.67&quot;</td>
</tr>
<tr>
<td>DARKKHAKI</td>
<td>&quot;0.74,0.72,0.42&quot;</td>
</tr>
<tr>
<td>BEIGE</td>
<td>&quot;0.96,0.96,0.86&quot;</td>
</tr>
<tr>
<td>LIGHTGOLDENRODYELLOW</td>
<td>&quot;0.98,0.98,0.82&quot;</td>
</tr>
<tr>
<td>OLIVE</td>
<td>&quot;0.50,0.50,0.0&quot;</td>
</tr>
<tr>
<td>YELLOW</td>
<td>&quot;1.00,1.00,0.0&quot;</td>
</tr>
<tr>
<td>LIGHTYELLOW</td>
<td>&quot;1.00,1.00,0.88&quot;</td>
</tr>
<tr>
<td>IVORY</td>
<td>&quot;1.00,1.00,0.94&quot;</td>
</tr>
<tr>
<td>OLIVEREDRAB</td>
<td>&quot;0.42,0.56,0.14&quot;</td>
</tr>
<tr>
<td>YELLOWGREEN</td>
<td>&quot;0.60,0.80,0.20&quot;</td>
</tr>
<tr>
<td>DARKOLIVEGREEN</td>
<td>&quot;0.33,0.42,0.18&quot;</td>
</tr>
<tr>
<td>Color Name</td>
<td>RGB Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>GREENYELLOW</td>
<td>&quot;0.68,1.00,0.18&quot;</td>
</tr>
<tr>
<td>LAWNGREEN</td>
<td>&quot;0.49,0.99,0.0&quot;</td>
</tr>
<tr>
<td>CHARTREUSE</td>
<td>&quot;0.50,1.00,0.0&quot;</td>
</tr>
<tr>
<td>DARKSEAGREEN</td>
<td>&quot;0.56,0.74,0.56&quot;</td>
</tr>
<tr>
<td>FORESTGREEN</td>
<td>&quot;0.13,0.55,0.13&quot;</td>
</tr>
<tr>
<td>LIMEGREEN</td>
<td>&quot;0.56,0.93,0.56&quot;</td>
</tr>
<tr>
<td>LIGHTGREEN</td>
<td>&quot;0.60,0.98,0.60&quot;</td>
</tr>
<tr>
<td>PALEGREEN</td>
<td>&quot;0.0,0.39,0.0&quot;</td>
</tr>
<tr>
<td>DARKGREEN</td>
<td>&quot;0.0,0.50,0.0&quot;</td>
</tr>
<tr>
<td>LIME</td>
<td>&quot;0.0,1.00,0.0&quot;</td>
</tr>
<tr>
<td>HONEYDEW</td>
<td>&quot;0.94,1.00,0.94&quot;</td>
</tr>
<tr>
<td>SEAGREEN</td>
<td>&quot;0.18,0.55,0.34&quot;</td>
</tr>
<tr>
<td>MEDIUMSEAGREEN</td>
<td>&quot;0.24,0.70,0.44&quot;</td>
</tr>
<tr>
<td>SPRINGGREEN</td>
<td>&quot;0.0,1.00,0.50&quot;</td>
</tr>
<tr>
<td>MINTCREAM</td>
<td>&quot;0.96,1.00,0.98&quot;</td>
</tr>
<tr>
<td>MEDIUMSPRINGGREEN</td>
<td>&quot;0.0,0.98,0.60&quot;</td>
</tr>
<tr>
<td>MEDIUMAQUAMARINE</td>
<td>&quot;0.40,0.80,0.67&quot;</td>
</tr>
<tr>
<td>AQUAMARINE</td>
<td>&quot;0.50,1.00,0.83&quot;</td>
</tr>
<tr>
<td>TURQUOISE</td>
<td>&quot;0.25,0.88,0.82&quot;</td>
</tr>
<tr>
<td>LIGHTSEAGREEN</td>
<td>&quot;0.13,0.70,0.67&quot;</td>
</tr>
<tr>
<td>MEDIUMTURQUOISE</td>
<td>&quot;0.28,0.82,0.80&quot;</td>
</tr>
<tr>
<td>DARKSLATEGREY</td>
<td>&quot;0.18,0.31,0.31&quot;</td>
</tr>
<tr>
<td>PALETURQUOISE</td>
<td>&quot;0.69,0.93,0.93&quot;</td>
</tr>
<tr>
<td>TEAL</td>
<td>&quot;0.0,0.50,0.50&quot;</td>
</tr>
<tr>
<td>DARKCYAN</td>
<td>&quot;0.0,0.55,0.55&quot;</td>
</tr>
<tr>
<td>AQUA</td>
<td>&quot;0.0,1.00,1.00&quot;</td>
</tr>
<tr>
<td>CYAN</td>
<td>&quot;0.0,1.00,1.00&quot;</td>
</tr>
<tr>
<td>LIGHTCYAN</td>
<td>&quot;0.88,1.00,1.00&quot;</td>
</tr>
<tr>
<td>AZURE</td>
<td>&quot;0.94,1.00,1.00&quot;</td>
</tr>
<tr>
<td>DARKTURQUOISE</td>
<td>&quot;0.0,0.81,0.82&quot;</td>
</tr>
<tr>
<td>CADETBLUE</td>
<td>&quot;0.37,0.62,0.63&quot;</td>
</tr>
<tr>
<td>POWDERBLUE</td>
<td>&quot;0.69,0.88,0.90&quot;</td>
</tr>
<tr>
<td>LIGHTBLUE</td>
<td>&quot;0.68,0.85,0.90&quot;</td>
</tr>
<tr>
<td>Color</td>
<td>RGB Color Codes</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>DEEPSKYBLUE</td>
<td>&quot;0.0,0.75,1.00&quot;</td>
</tr>
<tr>
<td>SKYBLUE</td>
<td>&quot;0.53,0.81,0.92&quot;</td>
</tr>
<tr>
<td>LIGHTSKYBLUE</td>
<td>&quot;0.53,0.81,0.98&quot;</td>
</tr>
<tr>
<td>STEELBLUE</td>
<td>&quot;0.27,0.51,0.71&quot;</td>
</tr>
<tr>
<td>ALICEBLUE</td>
<td>&quot;0.94,0.97,1.00&quot;</td>
</tr>
<tr>
<td>SLATEGREY</td>
<td>&quot;0.44,0.50,0.56&quot;</td>
</tr>
<tr>
<td>SLATEGREY</td>
<td>&quot;0.44,0.50,0.56&quot;</td>
</tr>
<tr>
<td>LIGHTSLATEGREY</td>
<td>&quot;0.47,0.53,0.60&quot;</td>
</tr>
<tr>
<td>LIGHTSLATEGREY</td>
<td>&quot;0.47,0.53,0.60&quot;</td>
</tr>
<tr>
<td>DODGERBLUE</td>
<td>&quot;0.12,0.56,1.00&quot;</td>
</tr>
<tr>
<td>LIGHTSTEELBLUE</td>
<td>&quot;0.69,0.77,0.87&quot;</td>
</tr>
<tr>
<td>CORNFLOWERBLUE</td>
<td>&quot;0.39,0.58,0.93&quot;</td>
</tr>
<tr>
<td>ROYALBLUE</td>
<td>&quot;0.25,0.41,0.88&quot;</td>
</tr>
<tr>
<td>MIDNIGHTBLUE</td>
<td>&quot;0.10,0.10,0.44&quot;</td>
</tr>
<tr>
<td>LAVENDER</td>
<td>&quot;0.90,0.90,0.98&quot;</td>
</tr>
<tr>
<td>NAVY</td>
<td>&quot;0.0,0.0,0.50&quot;</td>
</tr>
<tr>
<td>DARKBLUE</td>
<td>&quot;0.0,0.0,0.55&quot;</td>
</tr>
<tr>
<td>MEDIUMBLUE</td>
<td>&quot;0.0,0.0,0.80&quot;</td>
</tr>
<tr>
<td>BLUE</td>
<td>&quot;0.0,0.0,1.00&quot;</td>
</tr>
<tr>
<td>GHOSTWHITE</td>
<td>&quot;0.97,0.97,1.00&quot;</td>
</tr>
<tr>
<td>DARKSLATEBLUE</td>
<td>&quot;0.28,0.24,0.55&quot;</td>
</tr>
<tr>
<td>SLATEBLUE</td>
<td>&quot;0.42,0.35,0.80&quot;</td>
</tr>
<tr>
<td>MEDIUMSLATEBLUE</td>
<td>&quot;0.48,0.41,0.93&quot;</td>
</tr>
<tr>
<td>MEDIUMPURPLE</td>
<td>&quot;0.58,0.44,0.86&quot;</td>
</tr>
<tr>
<td>BLUEVIOLET</td>
<td>&quot;0.54,0.17,0.89&quot;</td>
</tr>
<tr>
<td>INDIGO</td>
<td>&quot;0.29,0.0,0.51&quot;</td>
</tr>
<tr>
<td>DARKORCHID</td>
<td>&quot;0.60,0.20,0.80&quot;</td>
</tr>
<tr>
<td>DARKVIOLET</td>
<td>&quot;0.58,0.0,0.83&quot;</td>
</tr>
<tr>
<td>MEDIUMMORCHID</td>
<td>&quot;0.73,0.33,0.83&quot;</td>
</tr>
<tr>
<td>THISTLE</td>
<td>&quot;0.85,0.75,0.85&quot;</td>
</tr>
<tr>
<td>PLUM</td>
<td>&quot;0.87,0.63,0.87&quot;</td>
</tr>
<tr>
<td>VIOLET</td>
<td>&quot;0.93,0.51,0.93&quot;</td>
</tr>
<tr>
<td>PURPLE</td>
<td>&quot;0.50,0.0,0.50&quot;</td>
</tr>
<tr>
<td>DARKMAGENTA</td>
<td>&quot;0.55,0.0,0.55&quot;</td>
</tr>
<tr>
<td>FUCHSIA</td>
<td>&quot;1.00,0.0,1.00&quot;</td>
</tr>
<tr>
<td>Color</td>
<td>RGB Value</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>MAGENTA</td>
<td>&quot;1.00,0.0,1.00&quot;</td>
</tr>
<tr>
<td>ORCHID</td>
<td>&quot;0.85,0.44,0.84&quot;</td>
</tr>
<tr>
<td>MEDIUMVIOLETRED</td>
<td>&quot;0.78,0.08,0.52&quot;</td>
</tr>
<tr>
<td>DEEPPINK</td>
<td>&quot;1.00,0.08,0.58&quot;</td>
</tr>
<tr>
<td>HOTPINK</td>
<td>&quot;1.00,0.41,0.71&quot;</td>
</tr>
<tr>
<td>PALEVIOLETRED</td>
<td>&quot;0.86,0.44,0.58&quot;</td>
</tr>
<tr>
<td>LAVENDERBLUSH</td>
<td>&quot;1.00,0.94,0.96&quot;</td>
</tr>
<tr>
<td>CRIMSON</td>
<td>&quot;0.86,0.08,0.24&quot;</td>
</tr>
<tr>
<td>PINK</td>
<td>&quot;1.00,0.75,0.80&quot;</td>
</tr>
<tr>
<td>LIGHTPINK</td>
<td>&quot;1.00,0.71,0.76&quot;</td>
</tr>
</tbody>
</table>