# TIBCO iWay® Service Manager

## Flow Debugger User's Guide

*Version 8.0 and Higher*
*March 2021*
*DN3502157.0321*

# *Contents*

Chapter 1

# Introducing the Flow Debugger

This section provides an introduction to the Flow Debugger.

**In this chapter:**



## Introducing the Flow Debugger

The Flow Debugger is a command line debugger for process flows deployed within a configuration. It runs in the iWay Service Manager console or in a telnet console. The user enters text commands and the debugger prints a text output.

The debugger allows for setting breakpoints and watchpoints, single-stepping nodes, debugging subflows, setting registers, changing the current document, among other functions. The debugger can be attached to a running application to debug in-situ if required.

When running the debugger, performance is slowed and memory usage is increased.

## Debugger Concepts

This section presents the concepts required to adequately use the debugger.

### Flow Development

A process flow is developed in iWay Integration Tools (iIT), and deployed to the iWay Service Manager (iSM).

### Operating Modes

The debugger works in two modes, standalone unit testing mode, or server configuration testing mode.

- **Unit Testing Mode.** In this mode, the debugger starts the flow. The input document comes from the debugger and the output document is trapped by the debugger. This mode is ideal if the flow can be tested without the channel protocol.

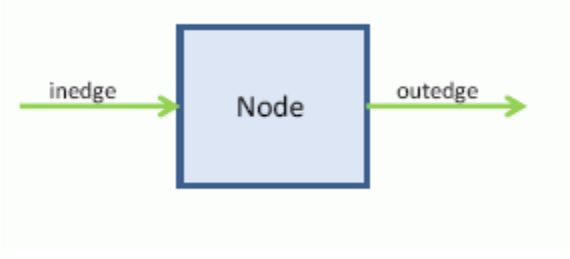
- **Configuration Testing Mode.** In this mode, the server channels start the flows and provide the initial documents. The output documents are handled normally by the channel outlets. This mode makes it possible to debug an entire application (iIA).

## Nodes and Edges

The flow consists of nodes connected by edges. Nodes are the units of execution in a flow.

The node name is specified in iIT. For more information regarding node names, see *Syntax* on page 25.

An edge coming into a node is called an *inedge*. An edge coming out of a node is called an *outedge*, as shown in the following example.




The same edge is both an *outedge* and an *inedge* for different nodes, as shown in the following example.


## Source

The source name tells the debugger where to find the source code of the process flow. For process flows deployed under a channel, the format is:

`<channelname>:[flowname]`

If *`flowname`* is not specified, then the process flow from the default route is selected. Note that *`channelname`* may contain colons internally. For example:

```
file1:testflow
```

```
channel1:inlet.1:File.1:testflow
```

For system flows, the format is:

```
flowname
```

where *flowname* cannot contain a colon.

For a process flow stored in a file, the format is:

```
[:]filepath
```

Without the initial colon, the *filepath* cannot contain a colon and must not collide with the name of a system flow. When present, the initial colon forces the rest of the value to be interpreted as a file path, which may include colons.

To list all of the process flows that currently exist, use the following command:

```
shell show flows
```

The debugger reads the compiled version of the flow. Small differences in the shape of the flow may be visibly compared to iIT. This is due to the flow compilation.

## Threads

A thread is a line of execution in a specific process flow. A thread may create other threads when execution splits into multiple edges, or when a subflow is called. All threads execute simultaneously, although some threads may be paused at a breakpoint.

The debugger is multi-threaded. It can control multiple threads within a flow and multiple independent flows simultaneously. While a thread runs, the debugger remains responsive to commands since it runs asynchronously with the flow.

The name of a thread has the format *W.channelname.workernumber.flowname[.split]* , where *split* can be absent. For example, the name of a thread running the flow *testflow* in channel *file1* with worker *3* might be *W.file1.3.flowtest*. In unit testing mode, the channel name is *debugger*, therefore the name of the thread would be *W.debugger.3.flowtest*.

For simplification, the debugger assigns a thread ID to each thread. The format is *tNNNN* , where *NNNN* is a unique monotonically increasing number. When printing the name of a thread, the debugger always precedes it with the ID.

For example, *t5: W.file1.3.flowtest*. In this example, the thread *W.file1.3.flowtest* has the id *t5*. The colon is a separator. It is not part of the ID.

In the command language, when a thread is specified, the thread name, or the ID, can be used interchangeably.

A thread terminates normally when it reaches an end node, or abnormally when it cannot handle an error condition. A process flow terminates when all its threads terminate. Therefore, before the original thread in the flow terminates, it must wait for all its child threads to terminate. The original thread is said to be in the waiting state when it has finished execution, but some child thread is still running.

The following table lists all the thread states:

| State | Description |
|---|---|
| running | Continuous execution. |
| waiting | Flow ended, waiting for child flows to end. |
| stepping | Executing a single node, skipping over subflows. |
| stepping_into | Executing a single node, entering into subflows. |
| stepping_nearest | Executing half a node, processing stage only, or dispatch stage only. |
| pausing | Suspended waiting for debugger command. |
| ending | Stopping execution per user's request. |

## Breakpoints and Watchpoints

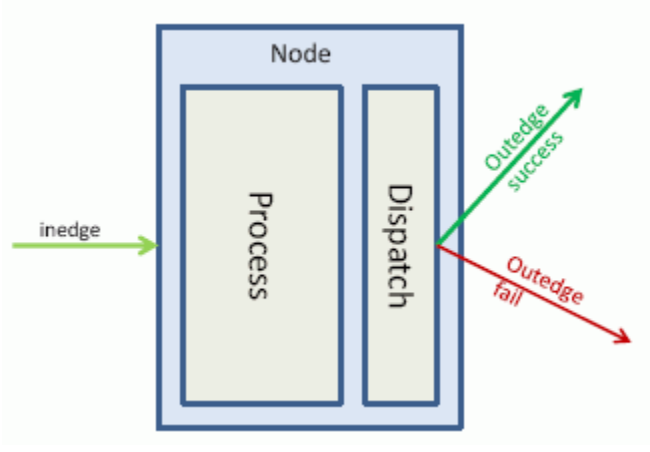
A breakpoint is an instruction to stop the execution of the flow at a specific location under some conditions.

A watchpoint is an instruction to stop the execution of the flow at any location when an event that satisfies some condition is detected.

Within a node, the execution is divided into two stages:

1. **The node is processed.** This returns a document and a list of edges.

2. **The document is dispatched on the matching wired outedges.** One outedge continues execution on the same thread. If multiple edges match, each remaining outedge forks a new thread.



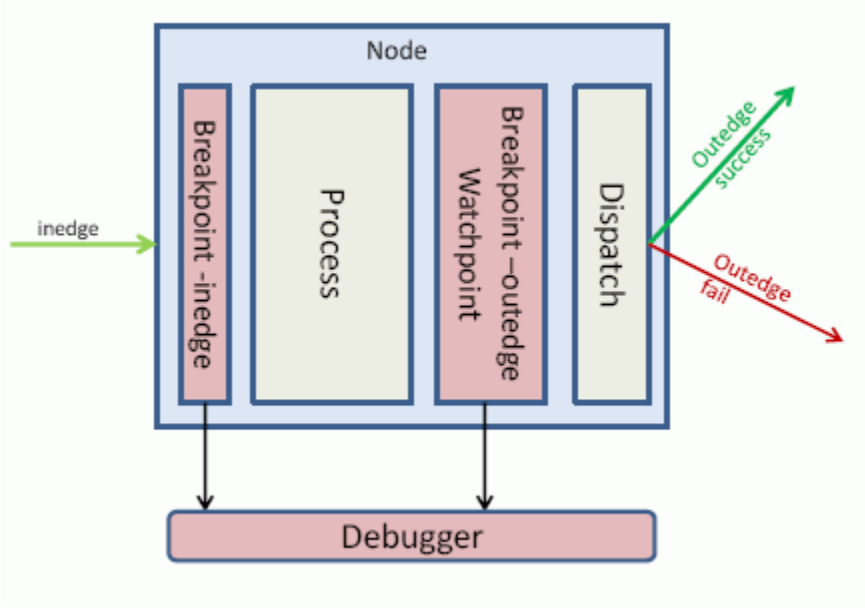

When none of the returned edges can be followed, dispatch will create a synthetic edge and attempt to match it. For example, it may attempt to match *OnDefault*, *OnCompletion* or *OnFailure*. If that still does not match, it will attempt *OnError*. If *OnError* is not followed and the error is not caught, the flow terminates abnormally.

The debugger supports two kinds of breakpoints depending on the stage where it stops within the node execution.

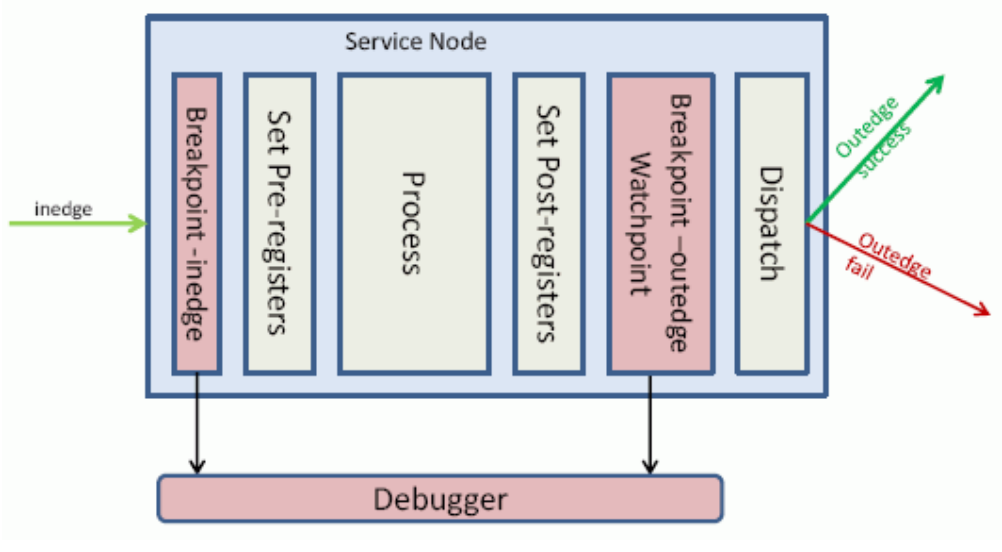
- **An inedge breakpoint.** Stops before the processing stage and therefore before the node executes. This is similar to a Java debugger that stops on a line before that line executes.
- **An outedge breakpoint.** Stops after the node processing but before the dispatch stage. There is no equivalent in a Java debugger.

A watchpoint behaves like an outedge breakpoint because it also stops before the dispatch stage. Logically, this identifies the node that satisfied the condition. This is more convenient than stopping at the start of the next node because there could be many nodes due to multiple outedges matching.

The following image depicts the *inedge* and *outedge* stages of the debugger:




For a service node, an *inedge* breakpoint occurs before the assignment of the pre-registers. Conversely, an *outedge* breakpoint occurs after the assignment of the post-registers.



It is possible to define one *inedge* breakpoint and/or one *outedge* breakpoint on a node, for a maximum of one at each location. If a duplicate breakpoint is defined at the same location but different conditions,it will overwrite the one previously configured.

Watchpoints are not tied to specific nodes. Nevertheless, if a duplicate watchpoint is defined for the same event but different conditions, it will overwrite the one previously configured for this event.

While paused on an *inedge* breakpoint, commands are available to inspect and modify: the *inedge*, the registers and the input document. The *inedge* is usually immaterial to the execution of the node except for iterator nodes.

While paused on an *outedge* breakpoint, possibly due to a watchpoint, commands are available to inspect and modify: the returned list of edges, the registers and the output document. The comma-separated list of edges has a significant impact on the flow execution as it will influence the result of the following dispatch stage.

The debugger assigns a unique breakpoint ID to each breakpoint. The format is *bNNNN* where *NNNN* is a monotonically increasing number. Watchpoints are also assigned a unique watchpoint id with the format *wNNNN*.

## Current Location

The debugger keeps track of the current source, the current node, and the current thread. The current node is always within the current source, but the current source and the current thread may be unrelated. For example, it is possible to set the current source to a subflow, before calling it from the current thread.

In general, the commands take their default arguments from the current location. Therefore, the current location is the default source, default node and the default thread.

The user can set the current location explicitly. The debugger also maintains the location intuitively.

## Special Registers

A special register holds a value. It is the equivalent of a variable for the flow. A register is defined in a scope. Scopes form a hierarchy from the most specific (at the thread level) to the most general (at the server level). A register is inherited from a higher scope unless it is redeclared at an inner scope. The inner scope register shadows the outer scope register, but both continue to exist. It is possible to access the outer scope register by specifying the correct scope. This is useful to share a value between threads, or to return a value to a higher scope before the current scope ends.

| Scope | Aliases | Description |
|---|---|---|
| local | thread | Most local scope for the current thread line. |
| flow | | Registers available throughout the flow. |
| message | global, system, worker | Registers available following completion of the flow. |
| channel | master | Registers available to all workers in the channel. |
| session | | Registers saved in the session. For example: cookies. |
| server | manager | Registers available to all channels in the configuration. |

Chapter 2

# Flow Debugger Operations

This section provides tutorials for debugger operations.

**In this chapter:**



## Performing Basic Operations in Unit Testing Mode

This tutorial demonstrates basic operation in unit testing mode, illustrating breakpoints, single stepping, setting the input document, and modifying registers.

**Note:** Though not illustrated in the tutorials, it is possible to abbreviate commands, tokens, and switches up to an unambiguous prefix. For example, the letter *b* is sufficient for the breakpoint command. A node name can also be abbreviated. It will be resolved among the nodes of the flow.

### *Procedure:* How to Perform Basic Operations in Debugger

For this tutorial, the process flow called *testflow* has already been deployed to channel file1. It stores the name of the root element in the special register *reg1* and tests whether *reg1* has the value *default*.

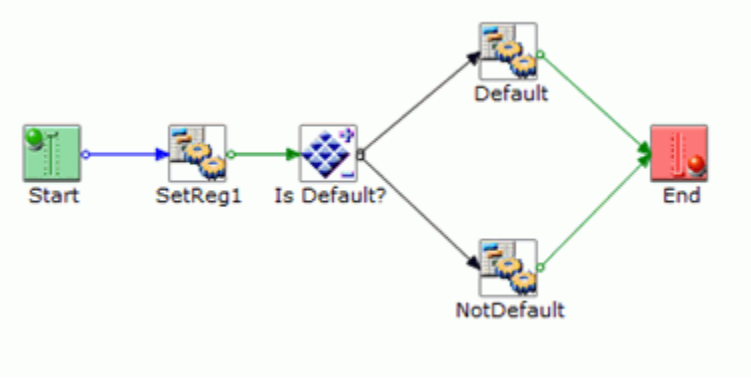
1. Start the debugger by typing the following command in the iSM console:

   ```
   Enter Command:> tool Debugger
   ```

   The debugger starts and responds with the *debug*> prompt, with the following default message:

   ```
   Command line process flow debugger
   Type help for more information
   debug>
   ```

The following image illustrates the process flow that is being referenced by this tutorial.




2. Change the default source to the source code of this flow as follows:

```
debug> set source file1:testflow
Current source changed to file1:testflow
```

3. List the nodes in the process flow and their relationships as follows:

```
debug> list
Node Start in source file1:testflow
  OnCompletion -> SetReg1
    OnSuccess -> 'Is Default?'
      true -> Default
        OnSuccess -> End
      false -> NotDefault
        OnSuccess -> End
```

The output has one line per node, beginning with the Start node. The outgoing edges are listed underneath, indented one level to the right. The format is *edgename -> nodename*.

For example, the second line says there is an *OnCompletion* edge from the Start node to the *SetReg1* node. In the subsequent lines, the *Is Default?* node has two out edges, true and false because they appear at the same indentation level.

You can use the list command to find out the exact names of the nodes.

## *Procedure:* How to Execute the First Run

1. Set a breakpoint on the *'Is Default?'* node. The name must be in quotes because it contains a space and a special character.

```
debug> breakpoint 'Is Default?'
breakpoint created
b1: breakpoint -source file1:testflow 'Is Default?' -inedge
```

2. Run the flow with the default document.

```
debug> run
Thread t1: W.debugger.1~file1:testflow started
Thread t1: W.debugger.1~file1:testflow suspended
by breakpoint at inedge of 'Is Default?'
```

The execution stops at the breakpoint.

3. Inspect the values of the special registers.

```
debug> show registers
Registers in pflow scope and above
   ibse-port = [CFG] '9000'
   iway.channel = [SYS] 'debugger'
   iway.config = [SYS] 'myapp'
   iway.flowname = [DOC] 'W.debugger.1~file1:testflow'
   iway.lastnode = [SYS] 'SetReg1'
   iway.pid = [SYS] '12748'
   iway.serverfullhost = [SYS] 'portable.ibi.com'
   iway.serverhost = [SYS] 'portable'
   iway.serverip = [SYS] '192.168.2.12'
   iway.startup.time = [SYS] '1401722478791'
   iway.workdir = [SYS] 'c:/iway/config/myapp'
   iwayconfig = [SYS] 'myapp'
   iwaydata = [SYS] 'c:/iway/'
   iwayhome = [SYS] 'c:/iway/'
   iwayversion = [SYS] '7.0.2'
   iwayworkdir = [SYS] 'c:/iway/config/myapp'
   name = [SYS] 'debugger'
   protocol = [SYS] 'Lcl'
   reg1 = [USR] 'default'
   tid = [DOC] 'b489cc73-ecf5-43b5-8cc5-9faab83cd972'
```

4. Execute a single step.

```
debug> step
Thread t1: W.debugger.1~file1:testflow suspended
at inedge of Default
```

The reg1 register contains the value *default*, therefore the execution steps to the Default node.

5. Resume execution.

```
debug> resume
Thread t1: W.debugger.1~file1:testflow resumed
Thread t1: W.debugger.1~file1:testflow terminated normally with 1
document
```

The flow reaches the End node and terminates.

6. Show the output document. For this flow, it is the same as the input document.

```
debug> show document
Document 0 from End
<default/>
```

### *Procedure:* How to Execute the Second Run

1. Run the flow again. Execution stops at the breakpoint.

   ```
   debug> run
   Thread t2: W.debugger.2~file1:testflow started
   Thread t2: W.debugger.2~file1:testflow suspended
   by breakpoint at inedge of 'Is Default?'
   ```

2. Set the value of the *reg1* register to *elem*.

   ```
   debug> set register reg1 elem
   ```

3. Execute a single step.

   ```
   debug> step
   Thread t2: W.debugger.2~file1:testflow suspended
   at inedge of NotDefault
   ```

   This time, the execution reaches the *NotDefault* node, since the register value is not *default*.

4. Resume execution. The flow terminates.

   ```
   debug> resume
   Thread t2: W.debugger.2~file1:testflow resumed
   Thread t2: W.debugger.2~file1:testflow terminated normally with 1
   document
   ```

### *Procedure:* How to Execute the Third Run

1. Delete the breakpoint.

   ```
   debug> delete b1
   deleted b1: breakpoint -source file1:testflow 'Is Default?' -
   inedge
   ```

2. Create a breakpoint on the *Default* node and another on the *NotDefault* node. Show the breakpoints.

   ```
   debug> breakpoint Default
   Breakpoint created
   b2: breakpoint -source file1:testflow Default -inedge

   debug> breakpoint NotDefault
   Breakpoint created
   b3: breakpoint -source file1:testflow NotDefault -inedge
   ```

```
debug> show breakpoints
b2: breakpoint -source file1:testflow Default -inedge
b3: breakpoint -source file1:testflow NotDefault -inedge
```

3. Execute the flow for the third time. Set the input document to change the root element name.

```
debug> run -xml <test/>
Thread t3: W.debugger.3~file1:testflow started
Thread t3: W.debugger.3~file1:testflow suspended
by breakpoint at inedge of NotDefault
```

The execution reaches the NotDefault node, since the reg1 register does not have the value *default*.

4. Show the value of the *reg1* register.

```
debug> show register reg1
reg1 = [USR] 'test'
```

5. Resume execution. The flow terminates.

```
debug> resume
Thread t3: W.debugger.3~file1:testflow resumed
Thread t3: W.debugger.3~file1:testflow terminated normally with 1
document
```

## *Procedure:* How to Execute the Fourth Run

1. Add an outedge breakpoint on the *'Is Default?'* node.

```
debug> breakpoint 'Is Default?' -outedge
breakpoint created
b4: breakpoint -source file1:testflow 'Is Default?' -outedge
```

2. Rerun the flow with the same arguments as the last run. In particular, the input document is <test/>.

```
debug> rerun
Thread t4: W.debugger.4~file1:testflow started
Thread t4: W.debugger.4~file1:testflow suspended
by breakpoint at outedge of 'Is Default?'
```

The execution stops after the 'Is Default?' node is processed but before the edges are dispatched.

3. Display the returned edges. The edge is false because the root element is not called default.

```
debug> show edges
Returned edge: false
```

4. Change the returned edge to true, and step to the next node.

```
debug> set edges true
debug> step
Thread t4: W.debugger.4~file1:testflow suspended
by breakpoint at inedge of Default
```

The execution reaches the Default node because the edge was changed to true.

5. End the debugger session. This will abort the execution of the flow before leaving the tool.

```
debug> end
Tool Debugger complete
```

## Using Flow Debugger in Configuration Testing Mode

This tutorial demonstrates the Flow Debugger in configuration testing mode.

### *Procedure:* How to Run the Debugger in Configuration Testing Mode

1. Start the debugger in configuration testing mode.

```
Enter Command:> tool Debugger -server
Command line process flow debugger [Server Mode]
Type help for more information
debug>
```

The process flows already being executed in the server are inaccessible, but every new instance of a process flow will be available to the debugger.

For this tutorial we assume the process flow called *testflow* from the first tutorial is still deployed to channel *file1*.

2. Verify *file1* is active by executing an iSM command without leaving the debugger.

```
debug> shell info
```

You should see information similar to the following image:

```
                                    completed failed  active  workers free
SOAP1
http          --active--  0             0       0       0       0
file          --active--  0             0       0       3       3
file1         --active--  0             0       0       3       3
```

3. If *file1* is stopped, start the channel with the following command: `shell start file1`

### *Procedure:* How to Execute the First Run

1. Change the default source to the flow in the default route of *file1*.

```
debug> set source file1:
Current source changed to file1:testflow
```

2. Set a watchpoint to trigger when the value of register *reg1* changes.

```
debug> watchpoint reg1
Watchpoint created
w1: watchpoint -source file1:testflow -scope local reg1
```

3. With a text editor, create a file called default.xml with the following content: `<default/>`
4. Copy this file to the input directory of *file1*.

   The channel picks up the file and runs the flow. The *SetReg1* node assigns the name of the root element to the *reg1* register.

```
Thread t1: W.file1.1~testflow suspended
by watchpoint at outedge of SetReg1
w1: watchpoint -source file1:testflow -scope local reg1
```

   The event was detected and the execution stops after *SetReg1* is processed, but before the returned edge is dispatched. The triggered watchpoint is displayed.

5. Show the *reg1* register to confirm the new value.

```
debug> show register reg1
reg1 = [USR] 'default'
```

6. Resume execution, the flow terminates.

```
debug> resume
Thread t1: W.file1.1~testflow resumed
Thread t1: W.file1.1~testflow terminated normally with 1 document
```

   Verify the presence of a file containing the output document of testflow, in the output directory of *file1*.

## *Procedure:* How to Execute the Second Run

Multiple threads can be debugged at the same time. This run shows two threads from the same channel, but it could be multiple threads from multiple channels. This makes it possible to debug an entire iIA application.

1. With a text editor, create a file called *test.xml* with this content:

```
<test/>
```

2. Copy both *default.xml* and *test.xml* to the input directory of *file1*.

   Since file1 has 3 workers, it will process both files concurrently. The watchpoint will trigger in both threads.

```
Thread t2: W.file1.1~testflow suspended
by watchpoint at outedge of SetReg1
w1: watchpoint -source file1:testflow -scope local reg1

Thread t3: W.file1.2~testflow suspended
by watchpoint at outedge of SetReg1
w1: watchpoint -source file1:testflow -scope local reg1
```

3. Show the threads.

```
debug> show threads
Threads
   t2: W.file1.1~testflow PAUSING at outedge of SetReg1
=> t3: W.file1.2~testflow PAUSING at outedge of SetReg1
```

4. Single step thread t3.

```
debug> step
Thread t3: W.file1.2~testflow suspended
at inedge of 'Is Default?'
```

5. Switch to the thread t2 and step.

```
debug> set thread t2
Current thread changed to t2: W.file1.1~testflow
debug> step
Thread t2: W.file1.1~testflow suspended
at inedge of 'Is Default?'
```

6. End the debugger session.

```
debug> end
Tool Debugger complete
```

This removes all breakpoints or watchpoints and resumes execution of all threads launched by the server. In this tutorial, t2 and t3 resume. Threads launched by the debugger are aborted. In this tutorial, there are no threads created by the debugger.

The flow continues until it terminates normally. The output document is handled by the outlet of file1. Verify the presence of two files containing the output document of the two threads in the output directory of *file1*.

Chapter 3

# Tips and Usage Considerations

This section provides a selection of tips and usage considerations for the Flow Debugger.

**In this chapter:**



## Documents

Nodes in a flow operate on a document, which can be displayed with the following command:

```
show document
```

This produces a large output if the document is large. The following command can be used to skip a prefix and only display a certain length.

```
show document -offset 10000 -maxlen 2000
```

If the payload is an XML document, it is preferable to evaluate an expression to extract the desired information as follows.

```
eval _XPATH(/INVOICE/INVOICE-NUMBER)
```

One method to change the payload of a document is to use the *Save* command to save it to a file, then edit the file and reload the payload before resuming operation, as shown below.

```
save document doc.xml
... edit doc.xml ...
set document -file doc.xml
```

## Sub-flows

Subflows are fully supported by the debugger. The iSM rule for a subflow is that it must end by returning a single document. The name of the End node of the subflow becomes the edge name followed by the calling service.

Subflows are deployed as system flows, independent of a specific channel.

You can step into a subflow if you are on the calling service. You can also set breakpoints and watchpoints in the subflow by first setting the source to the subflow.

If you are currently executing in a subflow and desire to simply complete the flow, a simple approach is to:

1. Set an outedge breakpoint on the calling service.
2. In the subflow, issue a resume.

## Event Flows

An event flow is executed when the server detects the corresponding event. For example, it is possible to configure the *Failed Reply To Flow*, *Dead Letter Flow*, and the *Channel Failure Flow* properties on a listener.

In unit testing mode, the debugger can launch these flows with a document supplied by the user. In configuration testing mode, the debugger can control the event flows executed by the server.

Event flows are deployed as system flows. Breakpoints and watchpoints can be set by first setting the source to the system flow, without the channel name.

## Testing a Portion of the Flow

Frequently, it is preferable to test a single node, or a small portion of the flow, against many conditions. Examples include testing all wired edges of a switch node, or testing the error handling of a catch node. The goto and set document commands make it possible to test a portion of a flow repeatedly without restarting the flow, as shown below.

```
goto node1
set document -file doc.xml
set register reg1 val1
step
show document
show registers
# repeat the commands to test node1 again
```

## Development Cycle

When a flow is modified, it needs to be redeployed. This will likely require exiting and restarting the debugger. The breakpoints and watchpoints can be preserved by saving the state before exiting. After restarting, they can be reloaded from the file, as shown below.

```
save state state.txt
end
... redeploy ...
tool Debugger
exec state.txt
```

## Messages

As a rule, the debugger does not follow messages through the server from channel to channel. A single message can, however, be followed by setting conditional breakpoints across the channels that test the TID special register. The TID, once assigned in the first channel to which the message is presented, remains with the message through its life in the application. While in a real production run the application may not change the TID, while debugging you can reset the TID special register to a simpler value to avoid tracking long, complicated character sequences. This process will simplify setting breakpoints and also not confuse other components of the server, such as iWay Business Activity Monitor (BAM).

**Note:** It is recommended that iWay BAM be disabled during a debugging sequence if the TID is to be changed.

```
breakpoint -if _sreg(tid)=='tidvalue'
```

Appendix A

# Flow Debugger Reference

This section provides reference information for the Flow Debugger.

**In this appendix:**



## Debugger Tool

Syntax: `tool Debugger [-server]`

The Flow Debugger can be started from the iSM console or a Telnet console. By default, the debugger starts in unit testing mode. When the -server switch is present, the debugger starts in configuration testing mode. This modifies the process flow engine to give the debugger access to any newly created flow, including flows executing as part of a channel. Only one server level debugger can be active at any one time in an iSM instance.

## Syntax

Command names, keywords and switches can be abbreviated to a minimum recognition, and are case insensitive. Items like channel names and flow names cannot be abbreviated, and are case sensitive. A node name can be abbreviated to an unambiguous prefix. It will be resolved among the nodes of the flow. Node names are case sensitive, but the case can often be ignored because of node name resolution.

If an item name contains a space or a special character, it must be quoted with single or double quote characters, like 'this' or "that". Lines beginning with # are considered comments and are ignored.

Tokens to be replaced are shown in <brackets>. Square brackets denote [optional] operands.Alternatives are separated with |.

The following table lists the replaceable tokens used in the syntax and their meaning:

| Token | Description |
|---|---|
| <channelname> | The name of a channel. For example:<br>`file1`<br>or<br>`channel1:inlet.1:File.1` |
| <command> | The name of a debugger command. |
| <count> | An integer. |
| <depth> | An integer. |
| <edge> | The name of a single edge. For example: OnSuccess, or fail_connect. |
| <edges> | A comma-separated list of one of more edges. For example: OnSuccess; or fail_redirection,303. |
| <expr> | An iFL expression. The debugger does not parse that part of the command. For example: `_XPATH(/INVOICE/INVOICE-NUMBER/DOC)=='555'` |
| <filename> | The path to a file. |
| <id> | A breakpoint ID of the form *bNNNN*, a watchpoint ID of the form *wNNNN*, or a thread ID of the form *tNNNN*, where *NNNN* is an integer. |
| <input> | The immediate data for an input document. The debugger does not parse that part of the command. For example: <root/> |
| <node> | The name of a node in the process flow as defined in iIT. The name can be abbreviated to a non-ambiguous prefix, and can be entered case insensitive. For example, "Node1" can be entered as "n" if there are no other nodes beginning with the letter "n". You will be alerted if the debugger detects ambiguity. If the entered name contains special characters (for example, a space as in "Node One"), then you must enclose the name in quotes. |

| Token | Description |
|---|---|
| <offset> | An integer for the number of characters or bytes to skip from the beginning. |
| <maxlen> | An integer for the maximum number of characters or bytes to process starting from the offset. |
| <reg> | A special register name. |
| <regpattern> | A pattern for a special register name. The character ? matches 1 character, and * matches 0 or more characters. For example: reg1 or ns.* |
| <scope> | A special register scope, valid values are: channel, flow, global, local, manager, master, message, session, server, system, thread, or worker. |
| <shellcommand> | A command for the iSM command executor. The debugger does not parse that part of the command. |
| <source> | Determines where to find the process flow source code. For process flows deployed under a channel, the format is:<br>*<channelname>*:[*flowname*]<br>If *flowname* is not specified, then the process flow from the default route is selected. Note that *channelname* may contain colons internally. For example:<br>`file1:testflow`<br>`channel1:inlet.1:File.1:testflow`<br>For system flows, the format is:<br>*flowname*<br>where *flowname* cannot contain a colon.<br>For a process flow stored in a file, the format is:<br>[:]*filepath*<br>Without the initial colon, the *filepath* cannot contain a colon and must not collide with the name of a system flow. When present, the initial colon forces the rest of the value to be interpreted as a file path, which may include colons. |

| Token | Description |
|---|---|
| <thread> | A thread ID of the form *tNNNN* where *NNNN* is an integer, or a thread name of the form `W.channelname.workernumber.flowname[.split]` where split can be absent |
| <type> | A register type. Valid values are: cfg, doc, hdr, met, sys, or user. Header registers are used to configure headers according to the protocol of the emitter. For other uses, the recommended type is user. |

## Commands

The following table provides a summary of all the available commands.

| Command | Description |
|---|---|
| breakpoint | Set a breakpoint on a node.<br>For more information, see *Breakpoint* on page 30. |
| delete | Remove a breakpoint or a watchpoint.<br>For more information, see *Delete* on page 31. |
| end | Terminate the debug session<br>For more information, see *End* on page 31. |
| eval | Evaluate an expression.<br>For more information, see *Eval* on page 31. |
| execute | Execute commands from a file.<br>For more information, see *Execute* on page 32. |
| goto | Transfer control to a specific node.<br>For more information, see *Goto* on page 32. |
| help | List commands or give help on specific command.<br>For more information, see *Help* on page 33. |

| Command | Description |
|---|---|
| list | Display flow anatomy.<br>For more information, see *List* on page 33. |
| rerun | Rerun the flow with the same parameters as the last run command.<br>For more information, see *Rerun* on page 34. |
| resume | Continue execution from the current location.<br>For more information, see *Resume* on page 34. |
| run | Run a flow.<br>For more information, see *Run* on page 34. |
| save | Save the debugger state or the contents of the current document to a file.<br>For more information, see *Save* on page 35. |
| set | Set a register, edge, current document, current source, or the current thread.<br>For more information, see *Set* on page 35. |
| shell | Execute an iWay Service Manager console command.<br>For more information, see *Shell* on page 37. |
| show | Display a value, such as threads, registers, etc.<br>For more information, see *Show* on page 37. |
| step | Execute a single node. With step into, start debugging a subflow.<br>For more information, see *Step* on page 39. |
| suspend | When the flow is running, suspend execution at the next node.<br>For more information, see *Suspend* on page 39. |

| Command | Description |
|---|---|
| watchpoint | Set a watchpoint to detect a change in register value, an edge returned or an edge about to be followed.<br>For more information, see *Watchpoint* on page 40. |

*Reference:* **Breakpoint**

Syntax: `breakpoint [-source <source>] <node> [-inedge|-outedge] [-count <count>|-if <expr>]`

A breakpoint is an instruction to suspend the execution of the flow at a specific node.

The source identifies which flow contains the node. The format is:

*channelname*`:|channelname:flowname | flowname |[:]<filepath>`

The default is the current source.

An inedge breakpoint stops before the node is executed. This is the default. An outedge breakpoint stops after the node has been processed but before the returned edges have been dispatched. For a service node, an inedge breakpoint suspends execution before the pre-registers are set and an outedge breakpoint suspends execution after the post-registers are set.

The count option suspends execution only after the breakpoint is reached that many times. When the breakpoint triggers, the count is reset to zero. This is useful for nodes in iterations.The conditional boolean expression suspends execution only if it returns true when evaluated. The expression has access to the current document and special registers.

The count option and the boolean expression cannot be used together. When the count and the expression are absent, the breakpoint suspends execution unconditionally.

With no arguments, the breakpoint command lists all defined breakpoints like *show breakpoints*.

The debugger assigns a breakpoint ID to the new breakpoint. The ID has the format *bNNNN* where *NNNN* is a unique monotonically increasing number. The ID can be used later to refer to the breakpoint, for example to delete it.

Examples:

- breakpoint Node1

- breakpoint Node1 -outedge
- breakpoint Node1 -count 10
- breakpoint Node1 -if _XPATH(/INVOICE/INVOICE-NUMBER/DOC)=='555'

### *Reference:* Delete

Syntax:

- `delete <id>`
- `delete breakpoint [-source <source>] <node> [-inedge|-outedge]`
- `delete watchpoint [-source <source>] [-scope <scope>] <reg>d`
- `delete watchpoint [-source <source>] -edge <edge>`

Removes a breakpoint or a watchpoint. Interactively, deleting with the ID is sufficient. In the longer forms, -inedge is the default and the default scope is local. These forms might be more convenient in a script (see the execute command).

Examples:

```
delete b1
delete breakpoint Node1 -outedge
delete watchpoint reg1
delete watchpoint -source file1:testflow -edge fail_connect
```

### *Reference:* End

Syntax: `end`

The debug session is terminated. Beforehand, the threads launched by the debugger are aborted. In configuration testing mode, breakpoints and watchpoints are removed, and paused threads originally launched by channels are resumed.

### *Reference:* Eval

Syntax: `eval [-thread <thread>] <expr>`

The eval command evaluates an iFL expression in the context of a thread. The default is the current thread. The evaluator has access to the current document and the special registers.

If no thread is specified and there is no current thread, then the expression must not require a document.

Examples:

```
eval _XPATH(/INVOICE/INVOICE-NUMBER)
eval _sreg(reg1)+_sreg(reg2)
```

## *Reference:* Execute

Syntax: `execute <filename>`

The file is used as a source of input commands. The commands are in the same format as commands entered at the command line. Blank lines and lines that start with # are considered comments and are ignored.

Files created by the save state command are compatible with the execute command. In this way the state of the debugger can be restored from an earlier session.

When preparing scripts expected to last across releases, it is wise to avoid using abbreviated syntax as new commands may later make the abbreviation ambiguous. Similarly, avoid use of specific IDs (threads, breakpoints…) as these may change from run to run.

Sample script:

```
# sample script
shell start file1
set source file1:testflow


breakpoint SetReg1 shell start iqueue1
set source iqueue1:qflow watchpoint -edge OnError
```

Example:

```
execute c:\script.txt
```

## *Reference:* Goto

Syntax: `goto <node> [-inedge|-outedge]`

Specify the next node to be executed in the current flow. The debugger will jump to the target node and pause, allowing an opportunity to set the document, registers, etc. By default, the debugger jumps to the inedge of the node and sets the input edge to OnSuccess. The *-outedge* option jumps to the outedge of the node without executing the processing stage. It also sets the returned edges to OnSuccess. The input edge or returned edges can be modified with the set command before execution is resumed.

Example:

```
goto SetReg1
goto 'Is Default?' -outedge
```

## *Reference:* Help

Syntax: `help [<command>]`

Give help on a specific command. List all commands if the command name is absent.

Examples:

```
help
help breakpoint
```

## *Reference:* List

Syntax: `list [<depth>] [-source <source>] [-node <node>]`

List the nodes and the edges between them. Specify the maximum depth displayed with an integer. This is useful for large process flows. The default is to display all nodes.

The source identifies which process flow is listed. The format is:

*channelname*`:|channelname:flowname | flowname|[:]<filepath>`

The default is the current source.

The listing starts at the identified node. If the source is specified, the default node is the start node, otherwise the default node is the current node within the current source.

The output has one line per node. The outgoing edges are listed underneath, indented one level to the right. The format is `edgename -> nodename.`

The debugger reads the compiled version of the flow. Some small differences in the shape of the flow may be visible compared to ilT. This is due to the flow compilation.

Sample Output:

```
Node Start in source file1:testflow
  OnCompletion -> SetReg1
    OnSuccess -> 'Is Default?'
      true -> Default
        OnSuccess -> End
      false -> NotDefault
        OnSuccess -> End
```

The second line says there is an OnCompletion edge from the Start node to the SetReg1 node. Lower down, the 'Is Default?' node has two out edges: true and false. That is because they appear at the same indentation level, which is one level to the right of 'Is Default?'.

Examples:

```
list
list 1
list -source file1:testflow -node 'Is Default?'
```

### *Reference:* Rerun

Syntax: `rerun`

Runs a flow using the same parameters as the last run command. The flow is rerun from the beginning but side effects of the flow, such as a file having been updated or a message sent to an external source, will not be rolled back. The rerun command uses the original input, even if the input came from a file and the file has been modified since then.

### *Reference:* Resume

Syntax: `resume [<thread>|-all]`

Execution continues from the breakpoint on the active or identified thread. The option -all causes all paused threads to be resumed.

Examples:

```
resume
resume t4
resume -all
```

### *Reference:* Run

Syntax: `run [[-source <source>] [-suspend] [-error] [[-bytes|-string|-xml] (-file <filename>|<input>)]`

Run a process flow. The source identifies the process flow. The format is:

*`channelname`*`:|channelname:flowname | flowname|[:]<filepath>`

The default is the current source.

The -suspend option causes the debugger to pause the flow before the Start node. By default, the flow executes normally.

The -error option marks the input document as in error. The default is not in error. The -bytes, -string, and -xml options specify the format of the input document. The default is XML format.

The contents of the document can be read from a file or directly on the command line. When not specified, the default document is <default/>. The encoding is UTF-8.

Examples:

```
run
run -source file1:testflow
run -file doc.xml
run -xml <root/>
run -string sample text
```

### *Reference:* Save

Syntax:

- `save document [-thread <thread>] <filename>`
- `save state <filename>`

Information is saved to a file for later use.

The save document command saves the payload of the identified document to a file. The current document of the specified thread is used. When omitted, the current document of the current thread is used. The output can later be used by the set document or run commands.

The save state command saves the current breakpoint and watchpoint settings into a file. The file can later be executed with the execute command to restore the state of the debugging session.

Examples:

```
save document doc.xml
save state breakpoints.txt
```

### *Reference:* Set

Syntax:

- `set alias <aliasname> <value>`
- `set document [-thread <thread>] [-error] [[-bytes|-string|-xml] (-file <filename>|<input>)]`
- `set edges [-thread <thread>] <edges>`
- `set inedge [-thread <thread>] <edge>`
- `set register [-thread <thread>] [-scope <scope>] [-type <type>] <reg> <expr>`
- `set source <source>`
- `set thread <thread>`

Set changes either a value or a debugger setting.

The `set alias` command creates a short name of a longer token that will be used frequently. For example, a channel name flowname combination such as `file1:inlet1:mainflow` might be given an alias of *mainflow*, as shown in the following example:

```
set alias mainflow file1:inlet1:mainflow
```

To refer to the longer name, use `$<alias>`, for example:

```
set source $mainflow
```

A good place to set aliases is in an executed script.

The `set document` command sets the current document on the identified thread. The default is the current thread. The -error option marks the input document as in error. The default is not in error. The -bytes, -string and -xml options specify the format of the input document. The default is XML format. The contents of the document can be read from a file or directly on the command line. When not specified, the default contents is <default/>. The encoding is UTF-8.

The `set edges` command sets the returned edges of the current node on the identified thread. The default is the current thread. The value is a comma-separated list of one of more edges. This command is enabled only when the thread is paused at the outedge of a node. The dispatch stage will match these returned edges against the outedges to determine which edges are followed.

The `set inedge` command sets the inedge of the current node on the identified thread. The default is the current thread. The value is a single edge name. This command is enabled only when the thread is paused at the inedge of a node. The inedge is usually immaterial to the execution of the node except for iterator nodes. The inedge called $jump means this is an iteration, any other name means it is the initial entry.

The `set register` command creates or modifies a special register. The register exists in the context of the identified thread. The default is the current thread. The default scope is local. The register type can be user, hdr or doc. The default is user. Header registers are serialized during emit operation to the proper header type for the protocol. Setting the register type applies during register creation only. An existing register will keep its register type. The reg token is the register name and the rest of the input is the iFL expression that will be evaluated before the result is assigned.

The `set source` command specifies the current source. The format is:

*`channelname`*`:|channelname:flowname | flowname|[:]<filepath>`

The default is the current source.

This becomes the default value for the source in other debugger commands.

The `set thread` command specifies the current thread. This becomes the default value for the thread in other debugger commands.

Examples:

```
set document -xml <root/>
set document -file doc.xml
set edges OnSuccess
set edges fail_redirection,303
set inedge $jump
set register reg1 _sreg(reg2)+3
set register -hdr User-Agent iWay Service Manager
set source file1:testflow
set source channel1:inlet.1:File.1:testflow
set source sysflow
set thread t4
```

## *Reference:* Shell

Syntax: `shell <shellcommand>`

Run a server command without leaving the debugger. The remainder of the command line is passed directly to the iSM command executor.

The commands are not interactive - that is, a command that depends on a prior command may not work as expected. Generally, the restrictions on the commands are the same as those that apply to commands executed by the schedule provider. See the console commands guide for further information.

Example:

```
shell set debug off -m file1
shell start file1
shell info
shell show flows
```

## *Reference:* Show

Syntax:

- `show`
- `show breakpoints`
- `show document [-attachments] [-noindent] [-offset <offset>] [-maxlen <maxlen>] [-thread <thread>]`
- `show edges [-thread <thread>]`
- `show node [[-source <source>] <node>]`

- `show registers [-thread <thread>] [-scope <scope>] [-inherit|-noinherit] [-type <type>] [<regpattern>|-hierarchy]`
- `show source`
- `show threads`

The show command without arguments displays the current location in the current thread.

The show breakpoint command displays the configured breakpoints.

The show document command displays the current document of the identified thread. The default is the current thread. If there are no active threads, the debugger will display the output document generated by the last flow started by the run or rerun command. By default, an XML document is pretty-printed. The -noindent option displays the XML document without extra indentation. The -offset option is an integer for the number of characters or bytes to skip from the beginning. The -maxlen option is an integer for the maximum number of characters or bytes to process starting from the offset. The attachments option displays the attachments associated with a multipart document.

The show edges command displays the inedge or the returned edges of the current node of the identified thread. The default is the current thread. The inedge is shown when paused on the inedge side of the node, or the returned edges when paused on the outedge side of the node.

The show node command displays the type, parameters and outedges of a specific node. The default is the current node of the current source. The format of the source is:

*channelname*:|channelname:flowname | flowname|[:]<filepath>

The show registers command displays the values of special registers accessible from the identified thread at the specified scope. The default is the current thread at the thread scope. If a register patternis entered, only registers with a name that match are shown. In the pattern, ? matches one character and * matches 0 or more characters. If the type is specified, only registers of that type are displayed. The -inherit option causes registers from higher scopes to be inherited unless they are redefined at a more local scope. This is the default and the usual behavior at runtime. The -noinherit option disables the inheritance and only shows registers defined locally within that scope. The scope can be one of: channel, flow, global, local, manager, master, message, session, server, system, thread, or worker. The -hierarchy option displays all the registers grouped by scope, starting from the specified scope. The default is -noinherit when the -hierarchy option is used.

The show source command displays the current source.

The show threads command displays the active threads. The arrow => indicates the current thread.

Examples:

```
show
show breakpoints
show document -noindent -offset 2000 -maxlen 500
show edges -thread t5
show node SetReg1
show registers
show registers reg1
show registers ns.*
show registers -scope worker
show registers -hierarchy
show source
show threads
```

## *Reference:* Step

Syntax: `step [-thread <thread>] [into|nearest]`

The step command executes the current node in the identified thread. The default is the current thread. If the thread is paused on the inedge of the node, both the processing and dispatching stages are executed. If the node is paused on the outedge of the node, only the dispatch is executed.

The step into command is identical to the step command except when the thread is paused on the inedge of a call to a subflow, in which case it will step into the subflow and pause on the start node instead. When paused on the outedge of the node (including a call to a subflow), the node is already processed, therefore step into is the same as step.

The step nearest command suspends at the nearest inedge or outedge location. If the thread is paused on the inedge of a node, it will process it and suspend at the outedge of the node. When paused at the outedge of a node, the step nearest command is identical to the step command. This command executes one stage of the node. It takes two step nearest commands to execute the node completely.This is particularly useful to process the node and inspect the returned edges before dispatch.

Examples:

```
step
step into
step nearest
step -thread t4
```

## *Reference:* Suspend

Syntax: `suspend [-thread <thread>] [-cancel]`

Suspend execution of the identified thread. The default is the current thread. The execution may stop at the outedge of the current node or the inedge of the next node. The thread can be resumed.

With the -cancel option, the debugger does a best effort to cancel the flow as soon as possible. The thread cannot be resumed. If the thread is in the running state, the debugger sends a cancel request to the current node. If the node is a service capable of cancellation, it will abort the processing immediately. If the thread is in the waiting state, the entire flow is cancelled, along with any subflows it might have been waiting for completion.

Examples:

```
suspend
suspend -cancel
suspend -thread t4
```

## *Reference:* Watchpoint

Syntax:

- `watchpoint [-source (<channel>|<source>)] [-scope <scope>] <reg> [-if <expr>]`
- `watchpoint [-source <source>] -edge <edge> [-followed] [-if <expr>]`

A watchpoint is an instruction to stop the execution of the flow at any location when an event that satisfies some condition is detected.

The first form declares a watchpoint that triggers when the value of a special register changes. The source and scope specifies where that register was created. The scope can be one of: channel, flow, global, local, manager, master, message, session, server, system, thread, or worker. The default is the current source and the local scope. The debugger will break on the outedge of the node that modified the special register. This node does not have to be in the same source, it could be in a subflow for example. In the case of a register at server scope, it could be any thread.

The second form declares a watchpoint that triggers when an edge is detected. By default, the watchpoint will match edges returned by the node, even if not followed. Optionally, it will match only edges that are actually followed.

The conditional boolean expression suspends execution only if it returns true when evaluated. The expression has access to the current document and special registers.

Examples:

```
watchpoint reg1
watchpoint -scope channel reg1
watchpoint reg1 -if _sreg(reg1)==100
watchpoint -edge OnError 11
```

Appendix B

# iSM Architecture Considerations for Flow Debugger

This section provides iSM architecture considerations for Flow Debugger.

**In this appendix:**



## iSM Architecture Considerations for Flow Debugger

Using the debugger effectively requires some understanding of iSM. This appendix is a non-technical discussion of iSM architecture and is limited to the basic knowledge required to debug.

iSM is a server that provides the ability to manage several channels each with an assigned protocol. For example, there may be a few file channels, a few HTTP channels, a few internal queue channels, etc. As each message arrives at the channel, it is assigned for work to a subchannel. The number of subchannels available to the channel is configured with the channel and can be set to vary depending upon workload.

In iSM, a channel is often called a master or listener; the subchannel is often called a *worker*.

The worker first encapsulates the incoming message into a document, which contains information about the message (has an error been encountered, are there attachments…). The message itself is called the payload, and it can be XML or non-XML. Additionally context information about the message (e.g. for a file channel the file name is stored) is made available in special registers. The document is passed to the process flow (Pflow) for business handling of the payload and context (registers). The debugger can manipulate the registers and document during the debug run as needed during the debug session.

Each worker is given an identifier, which takes the form W.channelname.n. One execution thread is assigned to each worker. This thread is passed into the PFlow, where additional execution threads may be assigned as the document takes simultaneous paths under control of the PFlow's program.

The Pflow program consists of nodes and edges. The nodes do the actual processing on the message and the edges connect the nodes one to the other. The debugger considers each node to have a lifecycle. This is shown in the diagram above in this manual. The inedge is the start of the node, when the interpreter first reaches it. The node process stage performs the actual task assigned to the node, such as doing a test or making an HTTP emit. Following this, the node reaches the outedge, where the registers being watched are checked for changes. The debugger then offers the opportunity (if an outedge breakpoint has been set) to see the edges that the process stage has selected to be followed. Passing control down these edges is called the dispatch stage. You can change the edge[s] to be dispatched if desired. The interpreter also supports synthetic edges which internally represent edges to be followed for specific conditions, such as $fail which matches any fail edge if no specific failure is taken.

iWay recommends the use of Integration Applications (iIA) to encapsulate a logical application. An iIA is an iSM runtime configuration. Although there is no strict rule that one configuration be one application, it is a common paradigm and in most cases a best practice. The debugger runs within the configuration and can be attached to the configuration's channels and Pflows.

In business applications, commonly multiple channels are employed in processing each message. A message can be passed from channel to channel under program (Pflow) control. Each channel often performs one step of the application processing. The debugger can be tasked to handle execution on multiple workers and multiple channels, enabling the configuration as a whole (the application) to be debugged.

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.



TIBCO, the TIBCO logo, the TIBCO O logo, FOCUS, iWay, Omni-Gen, Omni-HealthData, and WebFOCUS are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the readme file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.