# Interrupt Servicer

*Component Design Document*

## 1 Description

The Interrupt Servicer provides a task that an attached component executes on when an interrupt is received. The component is attached to an interrupt. When it receives the interrupt, the component's internal task is released, and the component calls the Interrupt_Data_Type_Send connector causing an attached component to execute on its internal task. A user-defined generic data type is passed between the Interrupt Servicer and the connected component. This data type will usually be filled in within a user defined custom interrupt handler, provided to the Interrupt Servicer at instatiation, and called by the Interrupt Servicer every time the specified interrupt is received.

## 2 Requirements

The requirements for the Interrupt Servicer component are specified below.

1. The component shall attach to a single, user-defined interrupt.
2. When an interrupt is received, the component shall pass a user-defined data type to a user-defined custom interrupt handler.
3. When an interrupt is received, the component shall pass the user-defined interrupt data to an attached external component.
4. The component shall contain an internal task on which an external component executes when an interrupt is received.

## 3 Design

### 3.1 At a Glance

Below is a list of useful parameters and statistics that give a quick look into the makeup of the component.

- **Execution** - *active*
- **Number of Connectors** - 2
- **Number of Invokee Connectors** - *None*
- **Number of Invoker Connectors** - 2
- **Number of Generic Connectors** - *None*
- **Number of Generic Types** - 2
- **Number of Unconstrained Arrayed Connectors** - *None*
- **Number of Commands** - *None*
- **Number of Parameters** - *None*

- **Number of Events** - *None*
- **Number of Faults** - *None*
- **Number of Data Products** - *None*
- **Number of Data Dependencies** - *None*
- **Number of Packets** - *None*

## 3.2 Diagram

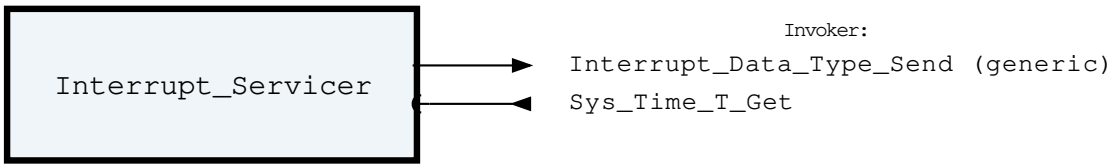


Figure 1: Interrupt Servicer component diagram.

The Interrupt Servicer has two connectors, one for sending a user defined (generic) data type that is filled in by the interrupt handler, and another for timestamping when the interrupt occurs. The type of the generic `Interrupt_Data_Type_Send` connector is determined by what data needs to be collected in the interrupt, and what data needs to be processed after an interrupt occurs.

The following diagram shows an example of how the Interrupt Servicer component might operate in the context of an assembly.

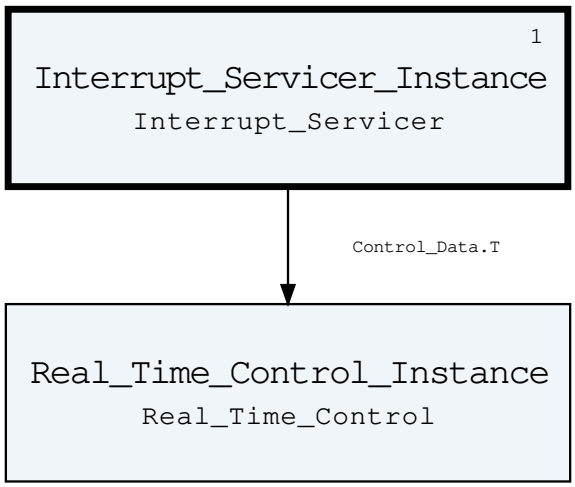


Figure 2: Example usage of the Interrupt Servicer in an assembly acting as the periodic trigger for a control law.

The Interrupt Servicer is attached to the Real Time Control instance which requires periodic control data in order to operate. When an interrupt is triggered, the user's custom interrupt handler is called which gather's the interrupt control data, storing it in the `Control_Data.T` record. The interrupt releases the Interrupt Servicer's internal task, which passes the `Control_Data.T` record to the Real Time Control component by invoking the `Interrupt_Data_Type_Send` connector. This effectively passes the execution context over to the Real Time Control component, which runs the control law using the newly captured data. After the Real Time Control component finishes execution, the Interrupt Servicer task waits until another interrupt is received. Upon the next interrupt, the whole process is completed again.

Note that in this assembly design, the Real Time Control component must execute quickly, so as to not drop any interrupts that might occur while it is executing. To avoid dropping interrupts, the Real Time Control component can be made active, and its `Control_Data_T_Recv_Sync` can be instead made asynchronous. In this way, interrupts received before the Real Time Control component has finished executing will be queued up for processing as soon as the Real Time Control component has finished processing the previous data.

## 3.3 Connectors

Below are tables listing the component's connectors.

### 3.3.1 Invokee Connectors

None

### 3.3.2 Invoker Connectors

The following is a list of the component's *invoker* connectors:

Table 1: Interrupt Servicer Invoker Connectors

| Name | Kind | Type | Return_Type | Count |
|---|---|---|---|---|
| `Interrupt_Data_Type_Send` | `send` | `Interrupt_Data_Type` (generic) | - | 1 |
| `Sys_Time_T_Get` | `get` | - | `Sys_Time.T` | 1 |

Connector Descriptions:

- **`Interrupt_Data_Type_Send`** - The data send connection.
- **`Sys_Time_T_Get`** - The system time is retrieved via this connector.

## 3.4 Interrupts

Below is a list of the interrupts for the Interrupt Servicer component.
Interrupt Descriptions:

- **`Interrupt`** - This component counts the number of times this interrupt occurs.

## 3.5 Initialization

Below are details on how the component should be initialized in an assembly.

### 3.5.1 Generic Component Instantiation

The Interrupt Servicer is parameterized by the Interrupt_Data_Type. The instantiation of this type is often determined by what data needs to be collected during an interrupt and how a connected component intends to process that data. This type a user defined type that is passed into the user's custom interrupt handler as an in-out parameter and then passed to a downstream components out of the Interrupt_Data_Type_Send connector. Usually, this type will be used to capture data related to an interrupt in the interrupt handler. Often, a timestamp, relating when the interrupt occurred, will be included in this custom type. The Interrupt Servicer will automatically insert the timestamp into the custom type if the second generic parameter, the Set_Interrupt_Data_Time procedure, is provided. This component contains generic formal types. These generic formal types must be instantiated with a valid actual type prior to component initialization. This is done by specifying types for the following generic formal parameters:

Table 2: Interrupt Servicer Generic Formal Types

| Name | Formal Type Definition |
|---|---|
| `Interrupt_Data_Type` | `type Interrupt_Data_Type is private;` |
| `Set_Interrupt_Data_Time` | `with procedure Set_Interrupt_Data_Time (Interrupt_Data : in out Interrupt_Data_Type; Time : in Sys_Time.T) is null;` |

Generic Formal Type Descriptions:

- **`Interrupt_Data_Type`** - The user's custom datatype that is is set in the custom interrupt handler and then passed to downstream components. If you do not foresee needing to collect specific data in the interrupt handler for the downstream component, consider using the Tick.T type, which includes a timestamp and a count.
- **`Set_Interrupt_Data_Time`** - This is an optional generic parameter that is used if the Interrupt_Data_Type includes a timestamp which should be filled in when an interrupt occurs. This parameter is a user defined procedure that, given a time, copies that time to the Interrupt_Data_Type. If this parameter is not provided, then time will not be set in the Interrupt_Data_Type. In this case, the user should not connect the Sys_Time_T_Get connector to avoid the CPU overhead of fetching a time that is never used. Note, if you are using Tick.T for the Interrupt_Data_Type, consider using the Handler procedure in the Tick_Interrupt_Handler package to instantiate this parameter.

### 3.5.2 Component Instantiation

This component contains the following instantiation parameters in its discriminant:

Table 3: Interrupt Servicer Instantiation Parameters

| Name | Type |
|---|---|
| `Custom_Interrupt_Procedure` | `Custom_Interrupt_Handler_Package. Interrupt_Procedure_Type` |
| `Interrupt_Id` | `Ada.Interrupts.Interrupt_Id` |
| `Interrupt_Priority` | `System.Interrupt_Priority` |

Parameter Descriptions:

- **`Custom_Interrupt_Procedure`** - A custom procedure to be called within the interrupt handler. The null procedure can be used here if no specific behavior is desired.
- **`Interrupt_Id`** - Interrupt identifier number for Interrupt
- **`Interrupt_Priority`** - Interrupt priority for Interrupt

### 3.5.3 Component Base Initialization

This component contains no base class initialization, meaning there is no `init_Base` subprogram for this component.

### 3.5.4 Component Set ID Bases

This component contains no commands, events, packets, faults or data products that need base indentifiers.

### 3.5.5 Component Map Data Dependencies

This component contains no data dependencies.

### 3.5.6 Component Implementation Initialization

This component contains no implementation class initialization, meaning there is no `init` subprogram for this component.

# 4 Unit Tests

The following section describes the unit test suites written to test the component.

## 4.1 *Tests* Test Suite

This is a unit test suite for the Interrupt Servicer component. It tests the component in a Linux environment, responding to signals.

Test Descriptions:

- **`Test_Interrupt_Handling`** - This unit test sends many interrupts to the Interrupt Servicer component and expects it to produce ticks.

# 5 Appendix

## 5.1 Preamble

This component contains the following preamble code. This is inline Ada code included in the component model that is usually used to define types or instantiate generic packages used by the component. Preamble code is inserted as the top line of the component base package specification.

```
1 -- Define the custom interrupt handling package type
2 package Custom_Interrupt_Handler_Package is new Interrupt_Handlers
  ↪  (Interrupt_Data_Type);
```

## 5.2 Packed Types

The following section outlines any complex data types used in the component in alphabetical order. This includes packed records and packed arrays that might be used as connector types, command arguments, event parameters, etc..

### Sys_Time.T:

A record which holds a time stamp using GPS format including seconds and subseconds since epoch (1-5-1980 to 1-6-1980 midnight).

Table 4: Sys_Time Packed Record : 64 bits

| Name | Type | Range | Size (Bits) | Start Bit | End Bit |
|---|---|---|---|---|---|
| Seconds | Interfaces.<br>Unsigned_32 | 0 to 4294967295 | 32 | 0 | 31 |
| Subseconds | Interfaces.<br>Unsigned_32 | 0 to 4294967295 | 32 | 32 | 63 |

Field Descriptions:

- **`Seconds`** - The number of seconds elapsed since epoch.
- **`Subseconds`** - The number of 1/(2^32) sub-seconds.