The following summarizes the proposals that have been made related to events. There are two parts that haven't always been separated well in the various proposals. I will attempt to capture the intent of the proposal as well. The purpose of this is not to provide the full details of each proposal, but to provide sufficient information for making a vote on what syntax and semantics should be integrated into the SV 3.1 standard.

Summary:



# 1 Event enhancement

This first group of proposals is focused on how to enhance the event type to solve zero delay races between event control and event triggers. The first two are nearly identical: they both add a new persistent event type but enhance a different part of the language to allow synchronization. The third leaves the type and synchronization method alone, and enhances the trigger. The fourth leaves the type and the trigger alone, and enhances the synchronization.

## 1.1 Add persistent event type, enhance wait

### 1.1.1 event bit

This proposal adds a new type of event, declared as event bit. The use of **bit** in the declaration implies that the new event has a semi-persistent state: its triggered status. The only way to change the state is by using the event trigger, ->. The event trigger will cause the value to go to the triggered state. At the end of that time-step, the value is automatically reset.

### 1.1.2 wait() enhanced to take old event

The state of **event bit** is determined by level sensitive control (the **wait** statement). This is the same way a **bit** would be tested by a **wait** statement, except **event bit** can't be used in an expression.

Level sensitive control is enhanced to allow the use of the old event type (previously as syntax error) and the new persistent event. Although the old event type has no value, the wait will block until it is triggered. This is the same as @(old_style_event) for non-persistent events. Since the new "**event bit**" has a value, the wait will unblock when either: the persistent event is triggered, or the persistent event is in triggered state. In other words, the **wait** statement is only enhanced to handle the old non-persistent events, not the new persistent events.

The event control requires only one enhancement. When a persistent event is added to event control, only entering the triggered state will cause the event control to unblock. A transition from the triggered to the reset state for an **event bit** will not unblock the event control. In this respect it is more like an **event** than the other variable types that have multiple states.

## 1.2 Add persistent event type, enhance event control

### 1.2.1 event bit

This proposal adds a new type of **event**, declared as **event bit**. The value of this new event type is not accessible, but it does require a tool to keep the state of the trigger. It uses the same event trigger operator as non-persistent events. When the persistent event is triggered, the event will hold the triggered state until the end of the time-step. At the end of that time-step, the state is automatically reset.

### 1.2.2 Event control, '@', enhanced to use new "event bit"

Event control, **@**, is enhanced to allow the new persistent event in addition to the non-persistent event. The event control checks the state of a persistent event and if it is already in the triggered state, the event control will unblock. If the persistent event has not been triggered before activation of the event control, the behavior will be the same as with a non-persistent event. Conceptually, this gives non-zero duration (within the bounds of the time-step) to the event trigger.

Level sensitive control, the **wait** statement, would not require enhancement. It would continue to be an error to use a synchronization object **event** or **event bit** in level sensitive control.

## 1.3 Enhance trigger only: ->>

This proposal doesn't change the declaration of events or the synchronization statements. Event control, the @ operator, is still used to detect events and it is still an error to use an event (persistent or non-persistent) in level sensitive control. This proposal adds a new trigger operator that removes races for events in the same fashion as nonblocking assignments remove races from zero-delay RTL. The nonblocking trigger works the same as the existing trigger, except that the triggering of the event is postponed until the nonblocking event queue. It may be possible to

define this as an even later queue once the semantics committee has settled on what those queues are called.

### 1.4 Enhance synchronization only: .active

#### 1.4.1 Add .active method to event type (all types?)

This proposal doesn't change the way events are declared or triggered. Instead it adds a new method that can be used on the event data type (it also suggested that it could be used on *any* data type).

#### 1.4.2 Enhance synchronization

##### 1.4.2.1 Enhance event control, @

The method would require the tool to keep a history of changes on the event so event control would know if the event had already occurred. Once event control was activated, it would check to see if a change had already occurred on the event (or any type?) earlier in the time-step. There would be no observable value from the active method and it would be a syntax error to use it in any context other than event control (same as an event).

##### 1.4.2.2 Use level sensitive control, wait()

The method would return a value indicating a change on the event. This would be zero if the event had not already triggered and one if it had. The value could be used in expressions and, in particular, the **wait** statement. The event can't be used in the **wait** statement by itself – that would still be a syntax error.

## 2 Synchronization enhancement

The second group of proposals focuses on enhancements that provide shorthand for waiting for events.

### 2.1 Examples

The following examples should be considered for inclusion in 12.7 to clarify the intent of the added syntax.

#### 2.1.1 all (for section 12.7.1)

The following example shows the equivalent Verilog code required to wait for three events (persistent or non-persistent) to be triggered.

Version required if 1.1 accepted:

```
fork
  wait(ev1);
  wait(ev2);
  wait(ev3);
join
```

Version required if 1.2, 1.3, or 1.4 accepted:

```
fork
  @(ev1);
  @(ev2);
  @(ev3);
join
```

### 2.1.2 any (for section 12.7.2)

The following example shows the equivalent Verilog code required to wait for any of three events (persistent or non-persistent) to be triggered.

Version required if 1.1 accepted:

```
fork
  wait(ev1);
  wait(ev2);
  wait(ev3);
join_any
```

Version required if 1.2, 1.3 or 1.4 is accepted –existing syntax supports this with event control and the **or** operator

```
@(ev1 or ev2 or ev3);
```

### 2.1.3 order (for section 12.7.3)

The following example shows the equivalent Verilog code required to wait for three events (persistent or non-persistent) to be triggered in order. Note that the $error may not be exactly equivalent a runtime error. Also, if any other child processes have been started outside this code fragment, they could be killed by the “disable fork.”

Version required if 1.1 accepted:

```
begin
  wait(ev1);
  fork
    wait(ev2);
    wait(ev1) $error();
  join_any
  disable fork;
  fork
    wait(ev3);
    wait(ev1) $error();
```

```
    wait(ev2) $error();
  join_any
  disable fork;
end
```

Version required if 1.2, 1.3, or 1.4 accepted:

```
begin
  @(ev1);
  fork
    @(ev2);
    @(ev1) $error();
  join_any
  disable fork;
  fork
    @(ev3);
    @(ev1) $error();
    @(ev2) $error();
  join_any
  disable fork;
end
```

## 2.2 Syntax

### 2.2.1 $wait_xxx

This provides an extension to the wait statement that parallels the proposal at 1.1 above. It enhances the wait statement by adding built-in system tasks. It implies that these extensions are level sensitive in nature, but with the extension of allowing non-persistent events. There are three versions: $wait_all (see example 2.1.1), $wait_any (see example 2.1.2), and $wait_order (see example 2.1.3).

### 2.2.2 wait_xxx

This is the same proposal as 2.2.1, except that keywords are used instead of built-in system tasks. The keywords would be wait_all (see example 2.1.1), wait_any (see example 2.1.2), and wait_order (see example 2.1.3).

### 2.2.3 @(event1 *magic_operator* event2)

This provides an extension of the event control that is more suited to proposals 1.2, 1.3, and 1.4, since they do not enhance the wait statement to take events. The **and** operator would yield the same behavior as example 2.1.1. The **or** operator provides the ability to wait for any event so there is no need for new syntax. To support

waiting for ordered events, a new operator would be required. Regardless of the operator, the semantics would be equivalent to the example 2.1.3.

#### 2.2.3.1 *magic_operator* for ordered events

Potential operators:

```
->
-->
=>
=>>
;
before
```

Operator selection only relevant if proposal 2.2.3 is selected.