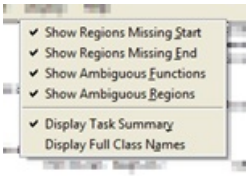
Mismatched Critical Regions



Potential issues involving the use of enable/disable interrupt calls or semaphores within tasks are detected by the Mismatched Critical Regions report along with the Calls in Critical Regions report. Both reports require that your critical region mechanism first be identified, as this is not explicit in C/C++ programs.

The Mismatched Critical Regions analysis identifies unmatched beginnings or ends of the critical (protected) regions. The report also locates areas where there is ambiguity about whether or not a given line of code is in a critical region. Such issues leave a software system vulnerable to unprotected interrupts, and may indicate other logic problems.

Critical regions are defined by specifying a starting and ending function call. For each starting function call found in the project, the report will check through all execution paths to the end of the program or task to find a matching ending function call. If some are missing, the report will indicate the starting call location and the closest path that is missing an ending call. In a second step, the report starts from each ending function call and examines all execution paths backwards from there to find matching starting function calls until it reaches the beginning of the program or task. If not found, the report will indicate the ending function call location and the closest path missing a starting call.

Ambiguity is considered to occur when the protected status of a line of source code depends on the path taken to that line. Note that the report will check all called or calling functions as part of this analysis. Examine the following example:

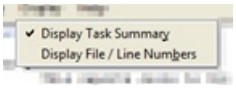
```
int globalX = 1;

void funcC() {
    int localC = 1;
    if (localC) {
        globalX = 1;
        EnableInt();
    } else {
        globalX = 2;
    }
}

void funcB() {
    int localB = 1;
    if (localB) {
 DisableInt();
        globalX = 1;
    } else {
        globalX = 2;
    }
}

void funcA() {
    funcB();
    funcC();
}


int globalY = 1;

void fc() {
  globalY = 2;
}

void fa() {
  DisableInt();
  EnableInt();
  fc();
}

void fb() {
  DisableInt();
  fc();
  EnableInt();
}
```

```
void ftop() {
  fa();
  fb();
}
```

In the above example, both the entry into and exit from the critical region are problematic, as only certain of the conditional paths are protected. The resulting report indicates both issues, and points to the closest path that is missing an entry or exit. This provides a staring point for reviewing the mismatches.

```
Mismatched Critical Regions

Settings:
        Critical Region:         CR ( DisableInt / EnableInt )

        Regions Missing Starts:  displayed
        Regions Missing Ends:    displayed
        Ambiguous Functions:     displayed
        Ambiguous Regions:       displayed

Task Definitions
Tasks are from Auto Task Generation: Any root functions
Name                Members  Graph  Root
autotask 1 - ftop         6    [+]  ftop
autotask 2 - funcA        5    [+]  funcA


Critical Region:  CR

Start of Critical Region                        Missing End of Critical Region
Function            Line File                   Function            Line File
funcB                 17 mismatched_cr.c        funcB                 21 mismatched_cr.c

Missing Start of Critical Region                End of Critical Region
Function            Line File                   Function            Line File
funcC                  7 mismatched_cr.c        funcC                  8 mismatched_cr.c

Ambiguous Functions
Function            Line File
fc                    32 mismatched_cr.c
funcC                  4 mismatched_cr.c

Ambiguous Regions
File                Lines
mismatched_cr.c     (all) 5 6 7 10 11 21 25 26 33
```

Event Transition between Tasks



Along with the Event Calls in Tasks report, the Event Transition Between Tasks report is used to check the use of events for synchronization in multi-tasking systems.

Both reports require that your event mechanism first be identified, as this is not explicit in C/C++ programs. Both the event functions and the events themselves are specified through the Event Definitions dialog. The events are those source code level identifiers (macros, enumeration literals, and constant variables) for the operating system shared resources that are used in event communications. The event functions are those operating system-specific functions used to wait for (pend) an event, to post an event (have the current task provide access to the shared resource), and to clear an event (remove any available postings for the event so that all tasks will again have to wait). The clear event function is assumed to be a variation of the post event function, where a special parameter value dictates that a clear rather than a post take place.

One major issue in inter-task synchronization is a deadlock, where every task is waiting forever for some other task to post an event. The Event Transition between Tasks report helps with identifying potential deadlocks and other issues. It serves as the starting point for browsing the program to verify the conditions under which tasks wait for each other.

Consider the following simple example where taskX signals taskY that it can now begin its activities:

```
#define EVENTA 1

extern void PostEvent(int event);
extern void WaitEvent(int event);

void taskX() {
   // gets started on interrupt, set things up
   PostEvent(EVENTA);
}

void taskY() {
   while (1) {
    WaitEvent(EVENTA);
    // do follow on work
   }
}
```

The report generated for this code indicates how EVENTA is used to communicate between the two tasks.

```
Event Transition Between Tasks

Key:
        P:              task posts this event
        W:              task waits for this event
        C:              task clears this event

Settings:
        Wait event function:       WaitEvent
        Post event function:       PostEvent
        Number of events defined:  1


Task Definitions
Tasks are from User Defined Tasks
Name        Members  Graph  Root
taskX             2    [+]  taskX
taskY             2    [+]  taskY

Events                                  Task1     Task2     Task3 ...

                                        taskX
                                          .       taskY
----------------------------------------  =         =
EVENTA ................................   P         W
```

Here's a more complex example, involving several tasks and using multiple events for signalling. The example also shows how the data flow analysis supports value propogation and expression

interpretation, along with some special case event posts.

```
// Events
#define EVENT1 1
#define EVENT2 2
#define EVENT3 4
#define EVENT4 8

void PostEvent(int event, int mode);
void WaitEvent(int event);

#define POSTMODE 1   // regular post
#define OS_FLAG_CLR 2  // clear posted events

void task1() {
    PostEvent(EVENT1, POSTMODE);
    // ignore because of OS_FLAG_CLR
    PostEvent(~(EVENT4|EVENT2), OS_FLAG_CLR);
}

void task2() {
    WaitEvent(EVENT1);
}

void task3_f1(int p) {
    int event;
    if (p == 13) event = EVENT2;
        else event = EVENT4;
    if (p) {
        PostEvent(event, POSTMODE);   // local var with direct assigns
    } else {
        WaitEvent(EVENT3);
    }
}

void task3() {
    int b = 2;
    if (b)
    while (b--)
        task3_f1(b);
    WaitEvent(EVENT2);
}

// more complicated event expression
int pevents = ~0;
void task4() {
    if (pevents & EVENT4)
        pevents &= ~EVENT1;
    else
        pevents &= ~(EVENT1 | EVENT3);
    PostEvent(pevents, POSTMODE);
    int wevents = EVENT1 | EVENT3 | EVENT4;
    if (pevents)
        wevents &= ~EVENT1;
    WaitEvent(wevents);
}

#define MULTI_EVENT1 (EVENT1 | EVENT3)
#define MULTI_EVENT2 (MULTI_EVENT1 | EVENT2)

void task5() {
    int c = 3;
    int event = EVENT4;
    if (c == 13) event |= MULTI_EVENT1;
    if (c) {
        PostEvent(event, POSTMODE);  // local variable with some expression
    }else {
        WaitEvent(MULTI_EVENT2);
    }
 }
```

```
// using arrays for event expressions
static const int execDefaultCallbackEvents[] =
{   // indexed by task id: must correspond to task id's defined in "tasks.h"
    EVENT1,           // SUPERVISOR task - default callback event
    NULL,                           // DISK       task - none
    NULL,                           // XFER       task - none
    NULL,                           // HOST       task - none
    NULL,                           // EXEC       task - none
    EVENT4              // BACKGROUND task - default callback event
};
void task6() {
    int taskid = 0;
    WaitEvent(execDefaultCallbackEvents[taskid]);
};
```

The resulting report shows, in matrix form, which tasks do posts, waits, and clears. In reviewing this, there are a couple of things to look for. Events (rows) that don't have at least one post call are obvious candidates for deadlock conditions. A more complex deadlock possibility occurs when two tasks (columns) are communicating using multiple events (rows), and the posts and waits occur in different tasks in each row. In this case, you'll want to check whether their control flow and conditions guarantee that the tasks don't wait for each other. You can use the Event Calls in Tasks report to browse to the individual calling locations of the waits to verify that no deadlock can happen.

```
Event Transition Between Tasks

Key:
        P:              task posts this event
        W:              task waits for this event
        C:              task clears this event

Settings:
        Post event function:       PostEvent
        Wait event function:       WaitEvent
        Number of events defined:  4


Task Definitions
Tasks are from User Defined Tasks
Name        Members  Graph  Root
Task1             2    [+]  task1
Task2             2    [+]  task2
Task3             4    [+]  task3
Task4             3    [+]  task4
Task5             3    [+]  task5
Task6             2    [+]  task6

Events                          Task1     Task2     Task3 ...

                                Task1
                                .      Task2
                                .         .      Task3
                                .         .         .      Task4
                                .         .         .         .      Task5
                                .         .         .         .         .      Task6
------------------------------  =         =         =         =         =         =
EVENT1 ......................   P         W         .         .        PW         W
EVENT2 ......................   P         .        PW         P         W         .
EVENT3 ......................   .         .         W         W        PW         .
EVENT4 ......................   P         .         P        PW         P         W
```

Note that one restriction of the event reports is that the event parameters used in the wait for and post event function calls need to be statically determinable. This means that the parameters need to be either the events themselves (macros, enumeration literals, or constant variables), simple expressions of the events (such as "EVENTA | EVENTB"), or local variables set to any of the events within the current function. The event parameters cannot be global variables or parameters passed into the current function.