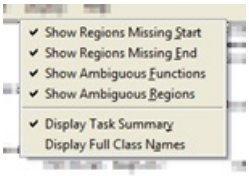
Mismatched Critical Regions



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:

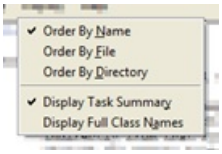
```
int globalX = 1;

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
```

Calls in Critical Regions



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

Functions that are called from within a critical region extend the time that other tasks are locked out. The Calls in Critical Regions analysis identifies all such functions so that they can be reviewed for appropriateness. Such calls from inside a critical region might also indicate an error in how the critical regions have been implemented. Consider the following code:

```
int getSensorData();

int input, output;

// start & end critical region
void DisableIrpt(), EnableIrpt();

void SetVars() {
    DisableIrpt();
    input = getSensorData();
}

void CalcOutput() {
    output = input * input / 0.25;
    EnableIrpt();
}

void Task1() {
    SetVars();

    PerformSomeOtherAction();

    CalcOutput();
}
```

The functions called from inside the critical region are identified in the report.

```
Calls in Critical Regions

Settings:
        Critical Region:  CR ( DisableIrpt / EnableIrpt )


Task Definitions
Tasks are from User Defined Tasks
Name        Members  Graph  Root
Task1             7    [+]  Task1

Calls in Critical Region                          File (Line)
          Task
             Line Number of Usage
                  Critical Region
                             User of Calls in Critical Region

CalcOutput                                        calls_in_cr.c (13)
          Task1
               23 (CR)     Task1                              calls_in_cr.c (18)

getSensorData                                     calls_in_cr.c (1)
          Task1
               10 (CR)     SetVars                            calls_in_cr.c (8)

PerformSomeOtherAction
          Task1
               21 (CR)     Task1                              calls_in_cr.c (18)
```