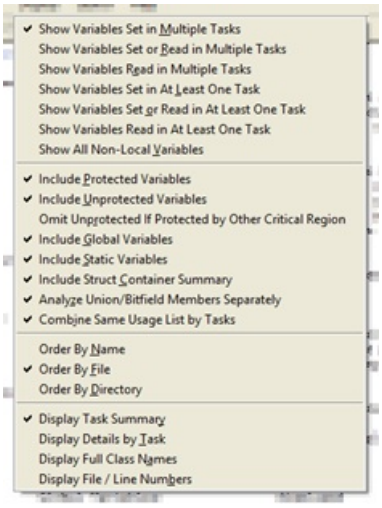
Variables Set in Multiple Tasks

Variables Set in Multiple Tasks reports global and static variables usage in multi-tasking software. As the report is highly configurable, you're able to have it filter for a variety of usage scenarios.



For example, you can have the report check for variables that are set in more than task. Such variable assignments can cause unforeseen and defective behavior in the tasks that use those variables if they are not protected by critical regions. The report supports five other specific scenarios. Alternatively, you can have the report list all usage of non-local variables, and review the use of each variable yourself. Consider the following code:

```
int globalA, globalB, globalC;

int subX(int paramX) {
    globalB = paramX;
    /* some calculations */
    paramX = globalB;
    return paramX;
}

void taskX() {
    int localX = 1;
    globalA = localX;
    /* some calculations */
    localX = globalA;
    DisableInt();
    localX = subX(localX);
    EnableInt();
    globalC = localX;
}

void taskY() {
    int localY = 1;
    globalA = localY;
    DisableInt();
    globalB = localY;
    EnableInt();
    localY = globalC;
}
```

Critical region protection, through the use of semaphores or interrupt control functions (DisableInt and EnableInt in this example) provides protection against unplanned interference with global variable values by other tasks. For example, the value of globalB when it's read in subX is assured of being the value assigned to globalB earlier in subX, as subX is protected by the DisableInt in taskX. As a result, an interrupt by taskY can't change globalB during the calculation time that occurs between the set and read of globalB in subX.

In contrast, the set and subsequent read of globalA in taskX is not protected. If an interrupt by taskY occurs in between the set and read, the value of globalA will be modified, interfering with the planned behavior of taskX.

The Variables Set in Multiple Tasks report includes analysis about whether the variable assignments occur in critical regions, and this information is included in the display. You're also able to filter out protected variable usage. Here, both protected and unprotected sets of variables are displayed. The access status for a given usage indicated by a U(nprotected) or P(rotected). For protected accesses, the name of the critical region indicated; here, one critical region has been defined in the Critical Region Definitions dialog, and given the name int.

```
Variables Set in Multiple Tasks

Settings:
        Critical Region:          int ( DisableInt / EnableInt )

        Usage Type:               set in multiple tasks
        Protected Variables:      displayed
        Unprotected Variables:    displayed
```

```
        Global Variables:          displayed
        Static Variables:          displayed
        Struct Container Summary:  omitted
        Union/Bitfield Members:    separate
        Combined Usage Listing:    on

Task Definitions
Tasks are from User Defined Tasks
Name        Members  Graph  Root
TaskX             4    [+]  taskX
TaskY             3    [+]  taskY

Variable                                          File (Line)
          Task
             Line Number of Usage
                  Critical Region
                              User of Variable

globalA                                           multi_set_vars.c (2)
          TaskX
               13 U   (int) taskX                           multi_set_vars.c (11)
          TaskY
               24 U   (int) taskY                           multi_set_vars.c (22)

globalB                                           multi_set_vars.c (2)
          TaskX
                5 P   (int) subX                            multi_set_vars.c (4)
          TaskY
               26 P   (int) taskY                           multi_set_vars.c (22)
```

One of the alternative usage settings of the report is to identify variables that have been used (set and/or read) in at least on of the tasks. As the listing of protected usage is omitted, the results focus on variable usage outside of the critical region protection. These uses could merit individual review.

```
Variables Used in Tasks

Settings:
        Critical Region:           int ( DisableInt / EnableInt )

        Usage Type:                set or read in at least one task
        Protected Variables:       omitted
        Unprotected Variables:     displayed
        Global Variables:          displayed
        Static Variables:          displayed
        Struct Container Summary:  omitted
        Union/Bitfield Members:    separate
        Combined Usage Listing:    on

Task Definitions
Tasks are from User Defined Tasks
Name        Members  Graph  Root
TaskX             4    [+]  taskX
TaskY             3    [+]  taskY

Variable                                          File (Line)
          Task
             Usage Type
                 Line Number of Usage
                       Critical Region
                                   User of Variable

globalA                                           multi_set_vars.c (2)
          TaskX
             S      13 U   (int) taskX                       multi_set_vars.c (11)
             R      15 U   (int) taskX                       multi_set_vars.c (11)
          TaskY
             S      24 U   (int) taskY                       multi_set_vars.c (22)


globalC                                           multi_set_vars.c (2)
```

```
          TaskX
             S      19 U   (int) taskX                      multi_set_vars.c (11)
          TaskY
             R      28 U   (int) taskY                      multi_set_vars.c (22)
```

When both sets and reads are analyzed, the usage type information about the accesses is added to the report. The access type is indicated by an S(et) or R(ead). Additional variables, such as globalC, will be listed, if they are only set in one task, but read in another.

Aggregate variables consist of structs and unions in C/C++ plus classes in C++. This report, along with some of the following ones, has options to control how aggregate variables are reported. If you ask for a Container Summary in the reports, then any member of these aggregates contributes to the container variable summary. For example, if one member of a struct is assigned in one task and another member of this struct is assigned in another task, then the members are not showing up in the Variables Set in Multiple Tasks report but the containing struct variable will.

Arrays on the other hand are considered a single variable by these reports because in general it is not possible to distinguish which array component has been assigned, as the indices can be dynamic expressions.

As much as is feasible via static analysis, Imagix 4D tracks which objects are referenced by pointers. Pointers or reference parameters track their actual parameter variables and derive potential changes that way. Pointers set to an array and used to index through an array cause the array to be modified. Imagix 4D assumes "clean" pointer usage, i.e. pointers are set to a certain variable and operate just on that variable but don't use side effects to access independent variables which are allocated in neighboring memory areas.