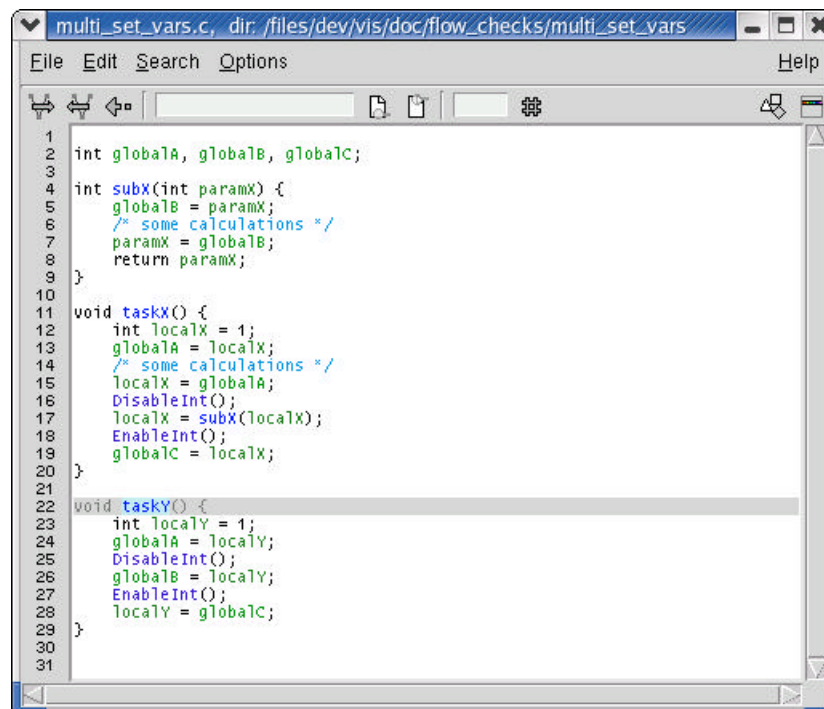


Variables Set In Multiple Tasks

Variables Set in Multiple Tasks reports any global and static variables used by more than one task. It might be considered a pair of reports. You can choose whether you want to review just set accesses, or set and/or read accesses.

The primary application of the report is to check just set accesses. Such variable assignments can cause unforeseen and defective behavior in the tasks that use those variables if they are not protected by critical regions.



```
multi_set_vars.c, dir: /files/dev/vis/doc/flow_checks/multi_set_vars
File Edit Search Options Help
1
2 int globalA, globalB, globalC;
3
4 int subX(int paramX) {
5     globalB = paramX;
6     /* some calculations */
7     paramX = globalB;
8     return paramX;
9 }
10
11 void taskX() {
12     int localX = 1;
13     globalA = localX;
14     /* some calculations */
15     localX = globalA;
16     DisableInt();
17     localX = subX(localX);
18     EnableInt();
19     globalC = localX;
20 }
21
22 void taskY() {
23     int localY = 1;
24     globalA = localY;
25     DisableInt();
26     globalB = localY;
27     EnableInt();
28     localY = globalC;
29 }
30
31
```

Critical region protection, through the use of semaphores or interrupt control functions, such as DisableInt and EnableInt here, 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 Define Critical Regions dialog, and given the name int.

```
Variables Set in Multiple Tasks
File Display Options Help

Variables Set in Multiple Tasks

Settings:
Critical Region:          int ( DisableInt / EnableInt )
Usage Type:              set only
Protected Variables:    displayed
Unprotected Variables:  displayed
Global Variables:       displayed
Static Variables:       displayed
Struct Container Summary: displayed
Union/Bitfield Members: separate

Task Definitions
Name      Members  Root
TaskX     4         taskX
TaskY     3         taskY

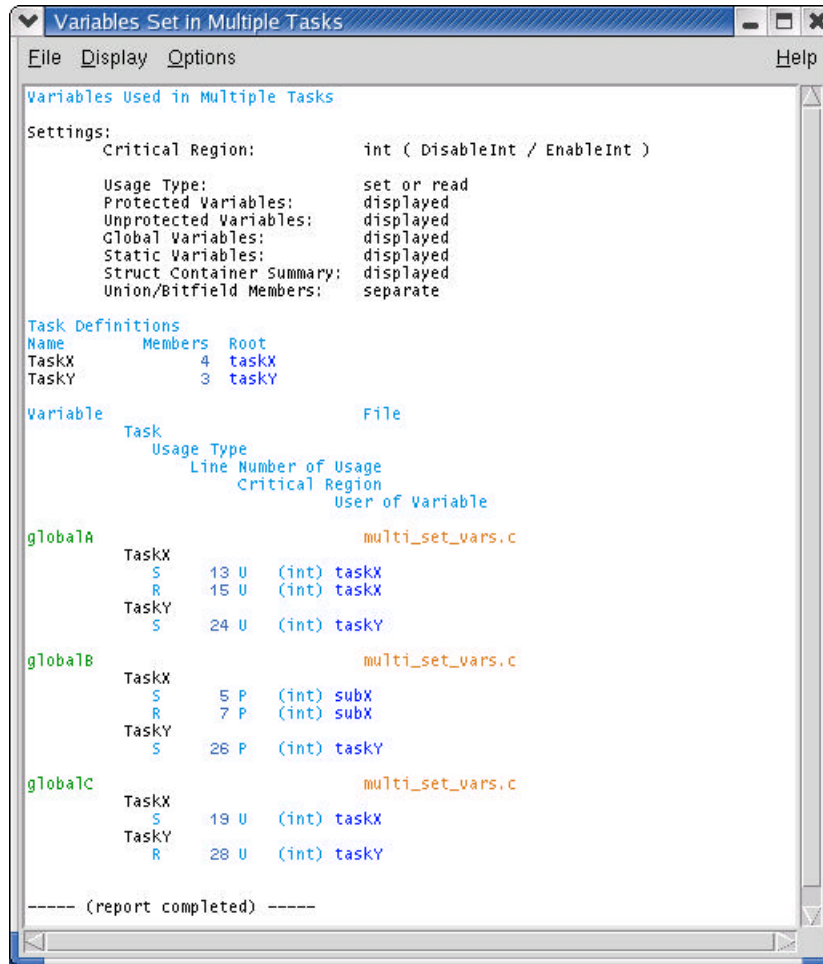
Variable      Task      File
Line Number of Usage
Critical Region
User of Variable

globalA
TaskX        13 U   (int) taskX
TaskY        24 U   (int) taskY

globalB
TaskX         5 P   (int) subX
TaskY        26 P   (int) taskY

----- (report completed) -----
```

The alternative application of the report is to analyze both set and read accesses of variables in multiple tasks. The results of this alternative analysis don't necessarily indicate a potential problem. Instead, they provide valuable overall insight into the usage and protection status of variables shared between tasks.



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 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 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.