IAR Application Note #17628

Interfacing visualSTATE and existing C code

The information in this document is based on version 4.2 of the IAR visualSTATE software. It may also apply to other versions of IAR visualSTATE.

SUMMARY

A visualSTATE design will often need to interface to existing C code written independently of visualSTATE. This independence of code may be intentional or not.

This document describes how to interface to functions and variables in such code.

KEYWORDS

interfacing, existing code, function, variable, visualSTATE-generated code.

The problem to be solved

A visualSTATE design will often need to interface to functions and variables defined in existing C code. The existing C code may intentionally have been written to be independent of any other pieces of code, or it may have been written prior to using visualSTATE.

In both cases, a visualSTATE design will often need to call existing functions, and to read and/or modify existing variables. The problem is how to accomplish this without modifying the existing code.

Solution

Using C, a visualSTATE design may need access to two different kinds of existing objects, i.e. functions and variables. Access to functions can often be accomplished quite easily, since visualSTATE is aware of externally defined functions. Access to variables is somewhat more difficult to accomplish, because visualSTATE will always define all variables itself. In both cases, the problem may become more complex due to the types involved.

Accessing externally defined functions

If an existing function must be called from a visualSTATE design, the declaration of the function must be included in the design as an action function. *Action function* is the visualSTATE term for a function declaration. When the action function has been created, the function can be called from within the visualSTATE design. The action function can either be typed in manually, or it can be imported automatically from a header file (see *IAR visualSTATE User Guide*).

Since the visualSTATE Coder generates function declarations based on the action functions in the visualSTATE design, it is crucial that the types of the return value and the function parameters match the types used in the definitions of the functions.

Normally a function is declared once in a single header file; the source file containing the definition of the function and all source files that contain calls to that function must include this header file. But since the Coder generates and uses its own function declarations, multiple declarations of the same function may occur. If these declarations are not identical, problems may occur. An example of this is shown in the following files:

```
existingCode.h
--------------
void func1 (unsigned char* c);

existingCode.c
--------------
#include "existingCode.h"

void func1 (unsigned char* c) {++*c;}

vsCode.c
--------
VS_VOID func1 (VS_UCHAR c);

void func2 (void) {func1 ('c');}
```

Note: The predefined type `VS_VOID` is equivalent to `void`, and `VS_UCHAR` is equivalent to `unsigned char`. To see the equivalents of the remaining predefined visualSTATE types, inspect the visualSTATE API file `vsTypes.h`.

By mistake, the user has made a wrong action function in the visualSTATE design, causing the visualSTATE Coder to generate a function declaration for `func1` that is inconsistent with the definition (the function accepts a pointer parameter, but the Coder-generated declaration accepts a value parameter). The compiler will not be able to discover that the function has different declarations, and unless the linker discovers this inconsistency, run-time errors may occur.

Since the types used for declaring functions in the visualSTATE design are limited to the predefined visualSTATE types, it can be impossible to match the types used in the definitions of existing functions. In such cases, wrapper functions should be used. For example, if an existing function `func` has a single parameter of type `short`, a wrapper function `wfunc` with a single function parameter of type `VS_INT` could be used:

```
void func(short i);

VS_VOID wfunc(VS_INT i) {func((short) i);}
```

Since an `int` (`VS_INT`) is guaranteed to be as large as a `short`, the type-cast does not cause problems as long as the values of the parameter used for calling `wfunc` is kept within the range of a `short`.

The problem with types becomes more complex if compound types such as unions and structs are to be used with existing functions. Pointers to objects of compound types can be passed through visualSTATE but they cannot be used within visualSTATE. A pointer to a compound type must be transferred to visualSTATE as an event parameter of type `void*` (`VS_VOIDPTR`), and must be passed further on to a wrapper function as a function parameter to an action function. The wrapper function must convert the pointer into a pointer of the correct type via a type-cast. If the compound object is defined with any kind of type qualifiers (`const`, `volatile`) or extended keywords, a type-cast may also be needed when passing the pointer into visualSTATE as an event parameter.

Accessing externally defined variables

visualSTATE will always define all variables that are to be used in a visualSTATE design. Thus the only straightforward way to access externally defined variables located in existing code is via functions. For every externally defined variable that must be read, define a function and declare that function as an action function in the visualSTATE design. For example, if `var` is an externally defined variable of type `int`, define the following function to read the value of the variable:

```
VS_INT ReadVar(VS_VOID) {return var;}
```

To modify the value of the variable, define another function:

```
VS_VOID SetVar(VS_INT value) {var = value;}
```

Workaround

There is a workaround that will allow a visualSTATE design to access externally defined variables directly, and not via functions. The workaround only works if all external variables are defined in existing code or other user-written code (*external variables* are the visualSTATE term for variables defined by visualSTATE that can be accessed both from within the visualSTATE design and the user-written code).

When using the Coder, make sure that visualSTATE Project and System external variables are generated in separate source file(s). For a description of visualSTATE Coder options, see *IAR visualSTATE User Guide*. The source file(s) will contain the definitions of all external variables. The workaround is to ignore these file(s), i.e. do not include them in the application project. Instead all variable definitions must be located in the user-written files.

Note that the definitions in the Coder-generated files may also include initialization of variables; if they do, make sure to initialize the variables in the user-written code, or enable and use the Coder-generated function(s) that initialize all external variables (refer to the chapter on Coder options in *IAR visualSTATE User Guide*).

Modifying existing code

Some of the approaches described here require the use of wrapper functions. If modifying existing code is an option, performance may be increased by making changes to the existing code.

Functions must still be defined externally to the visualSTATE design. However, where possible, change the types used in the declaration of the functions to the ones defined by visualSTATE in the file `vsTypes.h`. To minimize the risk of different declarations of the same function, include the Coder-generated file containing the function declarations in the user-written file that originally contained these function declarations.

Variables defined externally may be moved into the visualSTATE design as external variables. Such variables can be accessed both from within the visualSTATE design and the user-written code. Moving variables into the visualSTATE design causes a conceptual change of ownership of the variables: from being owned by the user-written code, the variables will be owned by the visualSTATE design. Such a change in ownership will increase performance.

Conclusions

In most cases it will be possible to interface to functions and variables in existing code from within the visualSTATE design. Interfacing to externally defined functions is fairly easy, while interfacing to externally defined variables is somewhat harder. In both cases wrapper functions may be needed for proper interfacing. If types are used that are different from the predefined visualSTATE types, interfacing becomes harder. If existing code can be modified, interfacing will be easier, and performance will be enhanced.

References

IAR visualSTATE User Guide.

Contact information

SWEDEN: IAR Systems AB
P.O. Box 23051, S-750 23 Uppsala
Tel: +46 18 16 78 00 / Fax: +46 18 16 78 38
Email: info@iar.se

USA: IAR Systems US HQ - West Coast
One Maritime Plaza, San Francisco, CA 94111
Tel: +1 415 765-5500 / Fax: +1 415 765-5503
Email: info@iar.com

USA: IAR Systems - East Coast
2 Mount Royal, Marlborough, MA 01752
Tel: +1 508 485-2692 / Fax: +1 508 485-9126
Email: info@iar.com

UK: IAR Systems Ltd
9 Spice Court, Ivory Square, London SW11 3UE
Tel: +44 20 7924 3334 / Fax: +44 20 7924 5341
Email: info@iarsys.co.uk

GERMANY: IAR Systems AG
Posthalterring 5, D-85599 Parsdorf
Tel: +49 89 900 690 80 / Fax: +49 89 900 690 81
Email: info@iar.de

DENMARK: IAR Systems A/S
Elkjærvej 30-32, DK-8230 Åbyhøj
Tel: +45 86 25 11 11 / Fax: +45 86 25 11 91
Email: info@iar.dk

March 2001.