ACSL to acslX Migration Guide

Version 3.0
March 2010

The AEgis Technologies Group, Inc.

410 Jan Davis Drive
Huntsville, AL 35806
U.S.A.
Phone: (256) 922-0802
info@acslx.com
www.acslx.com

ACSL to acslX Migration Guide

The AEgis Technologies Group, Inc.

410 Jan Davis Drive
Huntsville, AL 35806
U.S.A.
Phone: (256) 922-0802
info@acslx.com
www.acslx.com

March 2010

Table of Contents

Chapter 1 Introduction

acslX™ provides a single Integrated Development Environment (IDE) through which simulation model development, execution, experimentation, and results analysis are controlled. acslX includes both graphical and text-oriented modeling environments that give you full control over your models.

Functionally acslX includes the features from ACSL® 11.8 Sim, Graphical Modeller, and Math in one program. acslX builds on the main strengths of ACSL - speed, advanced solution algorithms, ability to solve non-linear problems, and a physical representation approach.

The following list highlights some of the new capabilities that acslX has over the ACSL 11.8 software:

- Comprehensive Integrated Development Environment (IDE) providing complete model design, development, execution, and analysis capability in one package
- Choice of target language between C/C++ and FORTRAN
- GNU C Compiler provided on distribution media
- Interactive debugging in both text and graphical modeling environments
- Dramatically improved Plotting Capability including 2-D, runtime, and 3-D
- Improved Export capabilities for results data and plots
- Advanced CSL Code Editor with context sensitive color coding
- Improved support for external C, FORTRAN, and M-Files
- Enhanced model API with C, .NET (C++, C#, and VB.NET) binding
- ACSL v11.8 GM porting tool
- Improved runtime prompt which incorporates the powerful analytical functions previously provided only with ACSL Math

This document describes specific procedures for migrating legacy ACSL 11.8 model source code and supporting files for use within acslX and acslXpress. Not all ACSL 11.8 models will require modification, but for the ones that do, an attempt has been made to provide a clear and easy migration path for CSL code, block diagrams, M-files and CMD files created for use with ACSL 11.8. It is strongly suggested that the appropriate portions of this guide and the release notes be reviewed prior to converting legacy models to acslX.

Chapter 2 Overview of Differences between ACSL 11.8 and acslX

In addition to the completely new Integrated Development Environment (IDE), many changes have been made to the CSL and Graphic Model translators, command interpreters, runtime libraries and generated simulation executable files.

In some cases, implementation of specific ACSL 11.8 language constructs or runtime commands has been deferred to an upcoming release of acslX; in other cases, specific legacy features have been removed entirely. The features that have been removed were those that did not fit well into the new technical architecture, where they were deemed to be of marginal usefulness, or where they historically promoted bad coding practices or errors in models. Furthermore, some of the file types used by ACSL 11.8 either have no counterpart in acslX, or have a corresponding file that is fundamentally different in format and content.

acslX no longer supports Watcom FORTRAN since Watcom it is no longer a commercially supported product. Compaq Visual FORTRAN continues to be supported along with the C/C++ compilers.

Some of the more important differences between ACSL 11.8 and acslX that impact the migration of legacy models are described in the following sections.

2.1 Changes to the User Interface

A primary goal of acslX is to provide an integrated user interface from which all activities associated with model development, execution, and analysis can be accomplished. To this end, the individual ACSL 11.8 applications have been consolidated into the acslX IDE. In addition to the changes in which application files are generated and stored in ACSL 11.8 versus acslX, this requires a certain level of familiarity with the new user interface in order to successfully migrate legacy models.

The acslX user interface and workflow structure introduces the concepts of workspaces and projects. A project is an organizational unit that collects all the information related to a particular model: a CSL file or Block Diagram, M-files, plot and result data, etc. A workspace is a collection of projects that are open simultaneously for editing, execution and analysis. To successfully port ACSL 11.8 simulations it is necessary to create appropriate workspaces and projects into which migrated legacy models or analysis scripts can be inserted.

The "Getting Started Guide" and the first few chapters of the "acslX User's Guide" provide detailed descriptions of the various components of the IDE and the basic workflow involved in setting up workspaces, projects and model files. Appendix A of the acslX User's Guide has many examples to gain familiarity with acslX.

2.2 Changes to the CSL Language

Minor changes have been made to the CSL language for use with acslX. In most cases, these changes resulted from restrictions imposed by the new support for multiple target languages; in particular, a number of language constructs which were FORTRAN-specific have been deprecated in the interest of promoting model code which is target-language neutral. A number of other structural and syntactic changes have been made to the CSL language in order to remove confusing language constructs, or to increase code readability and maintainability.

CSL keywords that are not used in acslX include the following:

CLOSE	INQUIRE	RANGE
COMMON	LINES	READ
EXPF	OPEN	WRITE
FORMAT	PRINT	

In addition, minor changes have been made to CSL syntax in the areas of white space, labels, comments, line continuation characters, implicit program definition, and argument list specifications. In addition, some system variables and utility routines present in ACSL 11.8 are not used.

Specific descriptions of changes, migration options, and examples are discussed in Chapter 3.

2.3 Differences between ACSL 11.X and acslX Block Diagrams

With a few exceptions, graphical models are constructed in acslX in a matter consistent with the way they were constructed in ACSL 11.8. acslX supports both single layer and compound PowerBlocks. Additionally, the default ACSL 11.8 block libraries (racks) have been ported to the acslX environment, and may be augmented with user-created block libraries in a manner similar to ACSL 11.8. Fundamental differences exist, however, in the way ACSL 11.8 and acslX store and translate graphical models.

ACSL 11.8 Graphic Modeler stores models in a binary format that is not readable, and directly generates CSL for use by the ACSL 11.8 Translator. In contrast, acslX stores block diagram information in standard XML notation; the acslX translator converts this XML file directly into target language source code without an intermediate CSL representation.

The fact that the graphical model files differ in format means that ACSL 11.8 graphical models are not directly reusable within acslX. Instead, legacy graphical models are ported to acslX file format using the ACSL 11.8 GM-Rack Converter Tool. This tool reads an ACSL 11.8 GM or Rack file as input, and generates a corresponding acslX block diagram or block library file that can then be inserted into an acslX model. Chapter 5 of this Guide describes the ACSL 11.8 GM and Rack transition process to acslX.

2.4 Changes to the ACSL SIM Command Language and ACSL Math Scripting Language

With the release of acslX v2.0 comes the introduction of a new command prompt (**>**) which combines the runtime control of ACSL 11.8 SIM with the powerful analytical commands previously provided with ACSL 11.8 Math.

With the exception of some plot-related commands, and commands related to low-level graphics primitives, nearly all ACSL 11.8 Math commands are supported in acslX without modification. Most ACSL 11.8 SIM runtime commands are supported, although the syntax of some commands may be different in acslX.

With this change, ACSL 11.8 CMD files need to be converted to m-files (.m files). acslX will convert existing CMD files to m-files automatically when they are added to a project.

A detailed list of unsupported and changed commands is provided in the following chapters. The acslX Command Reference provides detailed descriptions of modified command syntax.

2.5 Changes to the Simulation API

In ACSL 11.8, integration of a simulation executable with a user-generated host application is accomplished via the add-on module, ACSL Open API, which wraps generated simulation executables in a COM (also known as OLE2) API. This is a fairly straightforward way of integrating ACSL 11.8 simulations with applications developed in environments like Visual Basic or Visual C++, but precludes integration in any environment that does not support COM. In addition, COM adds a significant amount of computational overhead to applications that use it.

acslX provides a number of bindings to a generated simulation executable, which uses a simple C language callable API as its native interface. This interface can be called from any C or C++ code. In addition, a .Net binding is also provided which wraps this native API in a .Net class, which can be integrated with any language supported by the .Net platform. Primarily, this includes C#, Visual Basic .Net, and managed C++.

Details of the differences between the acslX API and the ACSL 11.8 API usage are discussed in Chapter 7. See also the acslX API Guide, located in Appendix B of the acslX User's Guide, for detailed information regarding integration of user code with generated acslX simulations.

Chapter 3 Migrating Legacy CSL Models

The following describes specific steps for migrating legacy ACSL 11.8 CSL models and supporting files. In general, the process consists of manually removing any reusable files from the ACSL 11.8 project and inserting them into a new acslX project, then making any necessary changes required for obsolete language constructs or new syntax. This section assumes familiarity with the files associated with ACSL 11.8 SIM.

- **Extract reusable files from the old project**. For projects built with ACSL Sim, CSL files, CMD files and any user-specified external libraries, object file, or target language source code files can be used by acslX.
- **Create a new acslX project and add the legacy files**. For details of creating an empty workspace and project, see the "Getting Started Guide" or Chapter 4 of the User's Guide.
- **Add an explicit program definition, if necessary.** While it was considered poor programming practice, ACSL 11.8 supported the notion of "implicitly" defined program structure: i.e., CSL programs did not require an encompassing PROGRAM/END section specification. This is now required in acslX, so open any applicable CSL file and make this addition, if necessary.
- **Remove obsolete constructs from the CSL file, if necessary**. Obsolete CSL keywords and syntax are described in detail below. It may be necessary to translate the model a few times to discover all of the places where edits will be necessary.
- **Remove calls to unsupported intrinsic routines, and replace them with equivalents where appropriate**. These include file open and close, read and write type calls. Unsupported intrinsic routines and system variables are described in detail below.
- **Make any necessary changes to user-defined code blocks or calls to user-defined external routines**. Target language code inserted into the CSL file must now be delimited with $TARGET_FILE/$END or $TARGET_FUNCTION/$END blocks. Where possible, the recommended alternative is to extract as much user-defined target language code as possible out of the CSL file, and place it into separate source code files which can be included in the acslX project and compiled along with the translated CSL file.
- **Make any necessary changes to CMD file(s)**. Add your existing .CMD files to your acslX project – they will automatically be converted to m-file scripts in order to work with acslX. Most of the ACSL 11.8 runtime commands are supported, and some may have a reduced syntax. All new analysis files should be created as m-files using the m-language syntax (See the command reference Manual).

3.1 Changes to the Macro Language

The Macro language used in ACSL 11.8 has been implemented in acslX with minor modifications. One subtle difference applies to the use of macro invocations within definitions of other macros. When a macro invocation is used, the arguments used in the macro call cannot include another macro invocation.

Instead, the argument that includes a macro invocation should be replaced with a dummy variable. Then create another line in the macro definition that assigns the argument to a dummy variable, which is declared using the REDEFINE macro statement. Moreover, it is recommended practice to not use complex expressions within macro call arguments.

For example, the following code should be modified as shown below:

Before:

```
QLDTX= SNGX*GFLX*ANX*SQRT(ABS(PLUX-PLDX))
```

After:

```
MACRO REDEFINE xtemp
xtemp = ABS(PLUX-PLDX)
QLDTX= SNGX*GFLX*ANX*SQRT(xtemp)
```

A second change to the macro syntax is the introduction of a new token (**#** instead of **&**) for the concatenation symbol (macro string substitution) – this was done to remove the ambiguity between *macro concatenation* and *line continuation* in macro definitions.

For example, the following code should be modified as shown below:

Before:

```
MACRO abc(x)
y = v&x&gh
MACRO END
```

After:

```
MACRO abc(x)
y = v#x#gh
MACRO END
```

3.2 GOTO Statement Inside an IF-THEN-ELSE Construct

Switching outside using a GOTO statement of an IF-THEN-ELSE construct is not permitted in acslX.

For example:

```
IF (XX.EQ.1.0) THEN
YY = XX
GOTO 10
ELSE
YY = 0.
ENDIF
10: CONTINUE
This should be programmed without the GOTO, as:
IF (XX.EQ.1.0) THEN
YY = XX
ELSE
YY = 0.
ENDIF
10: CONTINUE
```

3.3 Computed GOTO Statement

Computed GOTO statements are no longer allowed.

A computed GOTO statement in ACSL 11.8 is in the form:

```
GOTO (100,200), IVAL
```

Control is transferred to a statement labeled 100 or 200 if IVAL equals 1, or 2, respectively.

In acslX, this logic can be duplicated using a more structured approach as follows.

```
IF (IVAL .EQ. 1) THEN
GOTO 100
ELSE IF (IVAL .EQ. 2) THEN
GOTO 200
ENDIF
```

3.4 Discontinued CSL Constructs

This section describes details regarding general CSL language constructs that have been either changed or discontinued.

3.4.1 Implicit Program Definition

As described earlier in this chapter, ACSL 11.8 supported the notion of "implicitly" defined program structure, i.e., CSL programs did not require an encompassing PROGRAM/END section specification. Implicit program structure is not supported in acslX; all CSL programs must include an outer PROGRAM/END section definition.

3.4.2 Use of Whitespace

Use of whitespace in ACSL 11.8 CSL and GSL files was arbitrary. acslX allows the use of whitespace similar to the rules of conventional C-style programming languages. In particular, whitespace may not be inserted into variable names or subroutine/function names.

Also, commas must be used to separate items. For example, the following is not supported in an acslX compatible CSL or GSL file:

```
CONSTANT K1=1.0   K2=2.0   K3=3.0
```

It needs to be:

```
CONSTANT K1=1.0,   K2=2.0,   K3=3.0
```

3.4.3 Label Specifier

Support for old-style CSL label specifications using two periods (i.e., "<labelname>..") has been discontinued. All statement labels in CSL files must now use the colon delimiter ("<labelname>:").

For example:

```
10..CONTINUE
```

Is now

```
10:CONTINUE
```

3.4.4 Line Continuation Character

Use of the ellipsis ("...") to denote line continuation has been discontinued. Instead, use the ampersand "&" to denote continuations of lines of CSL code.

For example:

```
var6 =  var1 + var2 +var3 ...
+ var4 + var5
```

Is now:

```
var6 =  var1 + var2 +var3 &
+ var4 + var5
```

3.4.5 Comment Character

Single quotes can no longer be used to denote CSL code comments. CSL source file comments must now prefaced by the "!" character; all text following the "!" character to the end of line will be treated as a comment.

For example:

```
' this is a comment '
```

Is now

```
! this is a comment
```

3.4.6 Multiple Named Derivative Sections

Multiple unique derivative sections will not be supported in the first release of acslX. This functionality was allowed in ACSL 11.8 but was highly discouraged. However, it will be implemented in the future to allow multithreading of simulation models for parallelization requirements. Any derivative sections specified in the CSL file must be considered as parts of a single "default" derivative section; for translation purposes, they will be collected into a single section and sorted and translated as a unit.

3.4.7 Use of the "=" Operator in Argument Lists

The ACSL 11.8 translator reordered arguments to subroutine calls in cases where the argument list included the "=" operator to specify input and output arguments. In particular, input arguments were moved to the beginning of the argument list in the generated code. In acslX, argument list ordering is preserved. If the actual argument list order required by the subroutine does not match the ordering necessary to indicate subroutine inputs/outputs for sorting purposes, the entire subroutine call may be placed into a procedural block, where the procedural block declaration matches the previous argument list to the subroutine call. For example, if there was a call to a macro or subroutine such as:

```
CALL XYZ (OUT1, OUT2, OUT3 = IN1, IN2, IN3)
```

Where IN1, IN2 and IN3 are inputs, and OUT1, OUT2 and OUT3 are outputs, and the actual subroutine looks like:

```
SUBROUTINE XYZ (IN1, IN2, IN3, OUT1, OUT2, OUT3)
```

The call to the subroutine using an equals sign is not allowed in acslX. In order to let the translator know which are inputs and outputs, it is necessary to nest the subroutine call in a Procedural, and modify the call as follows.

```
PROCEDURAL (OUT1, OUT2, OUT3 = IN1, IN2, IN3)
CALL XYZ (IN1, IN2, IN3, OUT1, OUT2, OUT3)
END ! PROCEDURAL
```

3.4.8 Reserved Words as Variable Names

Use of reserve words (i.e., CSL key words) as variable names is no longer supported in acslX.

3.4.9 Switch Statements

Use of logical variables followed by a single period (log.x = 5) will no longer be supported in switch statements. However, acslX supports using logical IF statements (if(log) x=5).

3.4.10 Hollerith Constants

Use of Hollerith constants for specifying string literals are not supported in acslX.

3.4.11 End of Line

Use of the dollar sign ($) to define the end of a line is no longer supported in acslX; semicolon (;) is now used in acslX to specify the end of the line.

For example:

```
YZ = Y*Z $ XYZ = X*YZ
```

Is now

```
YZ = Y*Z ; XYZ = X*YZ
```

3.5 Deprecated Conventions

This section describes coding conventions, which, while still supported, are discouraged from continued used. In some cases, these may lead to unexpected results or code that does not port correctly across different target languages.

3.5.1 Undeclared Variables

Any model variable that is not explicitly declared is assumed to be an algebraic variable of type DOUBLEPRECISION. In many cases, this does not cause any problems. However, for the sake of code readability and portability, explicit declarations are recommended for all model variables. State variables are, of course, still declared by an INTEG, INTVC, IMPLC, IMPVC, DELSC or DELVC statement. Finally, it is recommended practice to declare any model variables before they are used.

A case that is often overlooked is the index as used in a DO statement. For example:

```
DO 100 i =1,10
! statements here
10: CONTINUE
```

You must declare "i" as an integer, as in:

```
Integer i
DO 100 i =1,10
! statements here
10: CONTINUE
```

3.5.2 The SORT Statement

While still supported in acslX, use of the SORT statement is strongly discouraged. In particular, the use of this statement in multiple and/or nested sections is not straightforward, and unexpected consequences may result. For code clarity and maintainability, it is recommended that any statements placed in a non-sorted section (e.g., INITIAL, DISCRETE) be coded in explicitly sorted order by the programmer.

3.5.3 Numeric Literal Type Mismatch

Numeric literals should always use notation specific to the appropriate numeric type of the variable. For example, floating point literals should include a decimal point or an exponent. This is particularly important when literals are used as initializers for variables or as arguments to functions or subroutine calls, because compiler-specific mechanisms for coercing data types cannot always be relied upon.

For example, if X is defined as a real variable, then:

X=1 should be coded as X=1.0

Similarly, if "I" is defined as an integer, then I=1.0 would be incorrect use of a literal.

Also be aware of the data types associated with CSL declarative statements such as ALGORITHM, CINTERVAL, MATERVAL, etc. Use of an integer literal where a floating-point literal is expected will produce semantic errors upon translating in some cases. Use of a literal of the incorrect type can have negative consequences in calls to external routines, especially in mixed-language programming where data is being passed from FORTRAN to C/C++, for example. In these cases use of an integer literal in place of a floating-point literal may result in a value passed to C/C++ code which consists of 32 bits representing the integer literal followed by 32 bits of garbage data

ACSL 11.8 and prior versions were very forgiving of ill-defined literals, so you will need to take care to review legacy code to ensure the model is free of mismatched literals.

3.6 Unsupported Operators and Intrinsic Functions

The following section describes currently unsupported CSL operators, utility functions, and system variables. Refer to the acslX Language Reference Manual for detailed descriptions of the syntax and usage of CSL operators.

3.6.1 I/O Functions

To accommodate translation to multiple target languages, certain operators related to file I/O that were FORTRAN-specific are no longer supported. Specifically, statements using the following operators must be removed from legacy CSL files: INQUIRE, OPEN, CLOSE, PRINT, READ, WRITE, and FORMAT. At present, file I/O from within CSL must be accomplished by defining external subroutines or functions which can be called from within the CSL code, and using the native file I/O facilities of the target language.

For example:

```
OPEN(10,file='results.csv')
WRITE(10,99)
99:FORMAT(' time , ph, salt, solid')
WRITE(10,100)time, ph, salt, solid
100:FORMAT(F7.0,',',F6.3,',',F5.2,',',F5.2)
```

Would use the following call in the CSL code:

```
CALL RESULTS (time,ph,salt,solid)
```

The subroutine that would be appended to the CSL file would be:

```
SUBROUTINE RESULTS (time,ph,salt,solid)
OPEN(10,file='results.csv')
WRITE(10,99)
99:FORMAT(' time , ph, salt, solid')
WRITE(10,100)time, ph, salt, solid
100:FORMAT(F7.0,',',F6.3,',',F5.2,',',F5.2)
END
```

3.6.2 COMMON and EQUIVALENCE

The FORTRAN COMMON statement is no longer supported. In cases when large volumes of simulation data must be made visible to externally defined code, it is now necessary to use target-language specific mechanisms for passing simulation data.

For example, when the target language is FORTRAN, acslX places all model variables into a global COMMON block, which may be made visible to external FORTRAN code. When the target language is C/C++, model variables are placed into a C-language structure, which may be referenced as an external symbol from user-defined C code.

The FORTRAN EQUVALENCE statement is supported with the Compaq Visual FORTRAN compiler only. When other compilers are used, it will be necessary to manually convert the statement into an equivalent representation by use of Procedurals. Section 4.2 discusses this in more detail.

3.6.3 Unsupported System Variables and Commands

System variables related to plots and printed output are no longer supported in acslX. Specifically, variables with names of the form xxxPLT, xxxPPL, xxxCPL, xxxSPL, xxxFPL and xxxPRN are no longer placed in the model dictionary.

Integration control variables (of the form xxxITG) which are still supported include the following:

CIOITG	ECSITG	TSMITG	WXDITG
CJVITG	MXOITG	WEDITG	
CSSITG	NRWITG	WESITG	
DPSITG	TJNITG	WNDITG	

All other ACSL 11.8 system variables are no longer supported. These include FDEITG and NXEITG, LOGD, DCVPRN, TITLE, SMOOTH, LINEAR are not presently supported.

3.6.4 Unsupported Utility Routines

The following ACSL 11.8 utility routines are not presently supported in acslX: LISTD, SETI, SETR and WEM. Remaining ACSL 11.8 utility routines as described in the ACSL 11.8 Language Reference are still supported and may be used in any CSL code regardless of the target programming language.

3.6.5 Reserved Names

The following names are reserved in acslX: CINT, NINT, SIGN

Chapter 4 Integration with Legacy C and FORTRAN Code

The manner in which user-defined target language code is integrated with CSL has been modified significantly from ACSL 11.8. In general, changes have been necessitated by the various requirements of the different target programming languages now supported.

acslX provides facilities that permit inclusion of user-defined C/C++ and FORTRAN code into CSL and Block Diagram models. This can be accomplished by embedding source code within CSL programs and GSL code blocks, or by inclusion of compiled object (.obj) or library (.lib) files into the project tree structure.

Inclusion of C/C++ or FORTRAN code into CSL/GSL is accomplished via **$TARGET_FILE** and **$TARGET_FUNCTION** code blocks. Blocks of code placed in these sections are extracted out of the CSL/GSL file by the translator's preprocessor, and placed in separate target language files during translation. These files are then included back into the translator-generated target language file using the file inclusion mechanism specific to the particular target language (i.e., an "#include" directive for C/C++ or a INCLUDE statement for FORTRAN).

An important distinction between **$TARGET_FUNCTION** and **$TARGET FILE** is that code blocks specified with **$TARGET_FUNCTION/$END** pairs are included at the beginning of each routine in the generated target language file and code blocks specified with **$TARGET_FILE/$END** pairs are included only once at the beginning of the target language file.

$TARGET_FUNCTION code blocks must precede the PROGRAM statement at the top of a native CSL file and **$TARGET_FILE** code blocks must occur at the end of the CSL file and after the END statement. In the case of GSL files, acslX will automatically place the **$TARGET_FUNCTION** code at the correct location during the translation process.

Chapter 7 of the acslX User's Guide provides additional information and examples on the integration of native C/C++ and FORTRAN code into acslX models.

4.1 Changes to User-variable Common Blocks

As noted above, the structure of the simulation code generated by acslX differs greatly from that generated by ACSL 11.8. For the most part, this has been necessitated by cross-language requirements arising from the need to use a common set of runtime simulation libraries with the various types of code now generated by the translator.

For example, consider the following CSL code, which demonstrates a method of accessing variables that are used in a separate Fortran subroutine, without having to pass the variables, constants, etc., by using formal parameters:

```
PROGRAM TESTCOMX
! Routine to test use of Global Common
! Statements in user supplied subroutines
INITIAL
CONSTANT xic = 0.5, xdic = 0.0
END

DYNAMIC
CINTERVAL cint = 0.2
CONSTANT tstp = 9.9

DERIVATIVE
CONSTANT  la = 0.6
xd      = INTEG(xdd, xdic)
x       = INTEG(xd, xic)
xdd     = la*(1.0-x**2)*xd - x

! CALL SUBROUTINE WITHOUT FORMAL PARAMETERS
! NEED TO DECLARE TYPE OF VARIABLE TO BE RETURNED
DOUBLEPRECISION X10
CALL TESTSUB
XOUT = X10 + 2.0
TERMT(t.GE.tstp, 'Stopped on Time Limit')
END ! DERIVATIVE
END ! DYNAMIC
END ! PROGRAM
$TARGET_FILE
SUBROUTINE TESTSUB
!     INCLUDE 'TESTCOM.INC'   (Used with ACSL 11.8)
USE GLOBAL_DATA
X10 = X*10.
```

```
END
$END
```

Examination of this model demonstrates how to utilize the "USE GLOBAL_DATA" command in a subroutine, in place of the "INCLUDE ModelName.INC" that was utilized in earlier versions of ACSL. Note the "DOUBLEPRECISION X10 " statement is required in the main program in order to let the software know what type of data is being passed.

Copying the above model and running it by using the following command file provides a simple test to verify the validity of this approach.

```
! TESTCOMX.m
prepare T X XD XDD X10 XOUT
output T X XD XDD X10 XOUT
start @nocallback
plot(_t,_x,_t,_x10,_t,_xout)
```

In particular, the numerous FORTRAN COMMON blocks that were used by ACSL 11.8 to organize the various model variables (both user and system) are no longer supported. For FORTRAN builds, all model variables are placed in a single COMMON block with the label "GLOBALS," which may be accessed by any user-specified FORTRAN code.

For builds that target the C language, all model variables are placed into a C structure. To integrate user code with a particular model, examine the generated code for that model; the structure, which contains all model variables, is defined at the top of the file. A single instance of this structure is also defined in the generated code. The variable associated with this instance may be called from external code by including a prototype for it at the top of the user-supplied file; this variable should be defined as "external".

4.2 INCLUDE Statements Must Contain Path Information

You can include Macros in a CSL model by using an INCLUDE statement. If the Macro is not in the project folder, the full path information must be supplied, as in:

```
INCLUDE 'C:\MyModel\MyMacros\MyMac.mac'
```

4.3 Including Legacy FORTRAN Code, Object Files, and Libraries

If there is Fortran 77 code that is to be compiled, the F90 compiler will handle it fine, but it will need to be compiled external to the model CSL code (i.e., not attached as shown previously). It must have a file extension of .FOR so the compiler will recognize it as F77 syntax. It must also be compiled with the same compiler switches as are used for the main model.

Use can be made of batch files run from the command prompt to accomplish this. Alternatively, the file can be associated with the main model by right clicking on the Model Files entry of the model tree view, finding the file (it needs to be in the project folder) and adding it to the project. Note that object files and libraries can be included in the project in the same way.

One restriction regarding files and libraries: if you link a library and an object file, the library cannot contain a copy of the object file, as the linker will not handle it properly. The same applies to having Fortran or C/C++ routines in the project with a duplicate named object in the library. The object will need to be extracted from the library.

In short, this means that existing libraries and routines can be used without modification. But all the source has to be recompiled using the same compiler switches as are used for the main model.

4.4 Table Sizes are Dynamically Allocated

Previous versions of ACSL required the user to set the Translator and Run-time Table sizes. In acslX this is not necessary; they are dynamically allocated.

4.5 Initializing Arrays

Suppose we have 3 arrays to initialize to a value of 0.0, and that the arrays are defined as:

```
Integer maxIngrs
Constant maxIngrs = 50
Dimension CompVals(maxIngrs,maxIngrs),
Results(maxIngrs,10,2)
Dimension Sens(MaxIngrs,10)
```

The following type of statements are not allowed:

```
CompVals=2500*0
Results=500*0
Sens=500*0
```

The following is the code needed to implement the above:

```
Integer kk, kkk, kkkk
DO 991 kk=1,maxIngrs,1
DO 990 kkk = 1,maxIngrs,1
CompVals(kk,kkk) = 0.0
990: Continue
991: Continue

DO 994 kk=1,maxIngrs,1
DO 993 kkk=1,10
DO 992 kkkk=1,2
Results(kk,kkk,kkkk) = 0.0
992:Continue
993:Continue
994:Continue

DO 996 kk=1,maxIngrs,1
DO 995 kkk=1,0
Sens(kk,kkk) = 0.0
995:Continue
996:Continue
```

4.6 DATA Statement changes

The Fortran DATA statement is not supported in CSL code since it has no counterpart in C code (it is actually a Fortran function). Therefore, DATA statements must be changed to use the CONSTANT statement.

For example:

```
DATA MyData / 1.0, 2.0, 3.0, 4.0 /
```

Needs to be modified to:

```
CONSTANT MyData = 1.0, 2.0, 3.0, 4.0
```

Initializing character arrays is done in similar fashion. For example:

```
Character Version*20
data version /'Mymodel July 17, 2002'/
```

becomes:

Character Version*20

```
constant version = 'Mymodel July 17, 2002'
```

When the desire is to use the data statement for access the /DATA flag for plotting, a new m command data was implement for use to load the data for use.

Syntax:

```
data(datablock_name, varname_array, data_matrix)
@clear
```

Description:

datablock_name = name of dataset

varname_array = column headers for the data being loaded

data_matrix = matrix containing the dataset

@clear = Use the “clear” flag to clear all previously defined data blocks

So the following ACSL 11.8 Sim Command directive…

```
DATA FOOBAR(T, X, Y)
1.2 3.4
2.3 4.5
?   6.7
8.9 ?
END
```

would be equivalent the data command from the command prompt or a script file:

```
data(“FOOBAR”, [“T”, “X”, “Y”], [0.0, 1.2, 3.4; 1.0,
2.3, 4.5; 2.0, NaN, 6.7; 3.0, 8.9, NaN]);
```

To utilize the data with the plot command you must use the !! to toggle to the old Sim command line. Follow the 11.8 format to use the data with the plot command.

Chapter 5 Porting Legacy Block Libraries ("Racks") and Graphic Models Using the Porting Tool

Although the format of graphical model and rack file differs between ACSL 11.8 and acslX, a porting tool has been provided to assist in the migration of these files. The porting tool will translate a majority of ACSL 11.8 GM constructs without additional user input, but certain models will require manual modifications to successfully translate and execute within acslX.

Procedures for operation of the porting tool, as well as descriptions of the types of manual edits that may be necessary, are described below.

5.1 Overview of Migration Procedure

This section assumes familiarity with the files associated with ACSL 11.8 Graphic Modeller. The general migration process consists of a few straightforward steps:

- Extract any reusable files from the project or rack. At present, the porting tool operates on GM, GSL and RAC files; generally, the porting tool will identify any supporting files associated with the GM file and will produce copies of them as a consequence of the porting process. Any CMD files associated with the project will have to be copied and migrated as described in the CSL file porting process previously.
- Run the porting tool.
- Move output files to desired location. Generally, this should correspond either to the location of the acslX project into which the converted model will be placed, or in the case of a rack conversion, to the location of the acslX block libraries.
- Add the converted files to the target acslX project. This applies to Graphic Models only. For details of creating an empty project and workspace, see the "Getting Started Guide" or the first few chapters of the User's Guide.
- Add the converted block library to acslX IDE. This applies to rack files only.
- Make any necessary manual changes. This applies to Graphic Models only. See the following sections for a description of the types of manual changes, which may be required for certain GM files.

5.2 Changes to the Translation Process

In addition to the changes which have been made to the graphical model and rack file formats, acslX introduces fundamental changes to the mechanism by which graphical models are translated into target language source code. In ACSL 11.8, the Graphical Modeller application persisted graphical models in a binary (non human-readable) format that was used exclusively by the GM application. The GM application also provided a mechanism for minimal parsing of CSL as required to construct variable lists associated with block code, and to perform construction of a conventional CSL file that was then passed to the ACSL 11.8 translator. The ACSL 11.8 translator was capable of translating CSL code only. In contrast, acslX uses the same XML-based format for persistence of block diagrams and for translation to target language code. The acslX translator includes capabilities for translating both conventional CSL model files, and the XML files generated by acslX graphical models into target language code. In the case of block diagrams, acslX does not require that the model be saved in an intermediate CSL representation.

5.3 Running the Porting Tool

A shortcut to the porting tool can be found on the start menu in the AEgis Technologies program group (**Start>Programs>AEgis Technologies>Tools**). For conversion of legacy models that use ACSL 11.8 Racks, use of this tool will require that ACSL 11.8 be installed on the system on which conversion is performed. Converted models include copies of the associated GSL files for any ACSL 11.8 Rack blocks, even if corresponding blocks exist in the acslX standard block libraries. For any blocks on the original diagram that include embedded or linked bitmaps, a copy of the bitmap is made by the conversion tool and placed in the output folder.

5.3.1 Specification of Input Files

For a GM file, click the "**Export GM Block Diagram to acslX**" button, locate the GM file in the file dialog, and click the "**Open**" button.

For a GM rack file, click the "**Export GM Rack to acslX**" button, locate the GM rack file in the file dialog, and click the "**Open**" button..

5.3.2 Conversion of GM Racks

What was referred to in 11.8 as a GM Rack is referred to as a block library in acslX. The porting tool can be used to convert GM racks to acslX block libraries. The porting tool simply converts the information from 11.8 into required format for acslX and creates any necessary supporting files. Chapter 7 of the acslX User's Guide describes the method for including block libraries into acslX.

5.3.3 Location of Output Files

After conversion, a "converted" folder is created in the same location as the file being ported. The new acslX block diagram (GMX) or block library (ARK) file will be located here, along with other supporting files such as images, GSL files, etc. All files in this directory are required for the newly converted block diagram or block library to operate properly.

5.3.4 Changes to Specific Block Types and Unsupported Model Constructs

For the most part, the acslX GM/Rack Conversion tool will port GM file or Racks with minimal manual user input. In certain cases, though, some editing of converted files will be required in order to ensure accurate conversion of the legacy model code. The specific cases, in which the requirement for manual modification of ported files is required, are described below.

5.3.5 Conversion of Non-unique Block Names

ACSL 11.8 required that block names be unique only within the scope of a particular compound block (or at the top level of the diagram). Unique variable names were constructed by prepending the names of any parent compound blocks in the hierarchy to the name of a particular variable in the CSL code associated with a block. acslX requires that block names be globally unique within a model. Hence, it may be the case that a particular block name in a legacy model will not be directly usable within acslX. In these cases the porting tool modifies the name of the offending block slightly to make it unique within the model. In general, this does not require any user intervention; the generated block names will be valid for translation purposes. If for some reason the generated names are aesthetically unacceptable, they may be changed from within acslX after the model has been ported.

5.3.6 Plot Blocks, Creation of User-defined M-code Blocks

Implementation of plot blocks with acslX differs significantly from that of ACSL 11.8. In acslX, plot blocks are really segments of M-code, which are executed when the block is double-clicked. Currently, ACSL 11.8 plot blocks are not ported; an empty block is put in the place of any plot block within an ACSL 11.8 graphical model. After conversion, these blocks must be removed and replaced with an appropriate plot block from the acslX plot block library. This is needed to incorporate the improved plotting functionality in acslX.

5.3.7 ACSL 11.8 C-code Blocks

At present, the acslX GM-Rack Conversion tool does not support conversion of ACSL 11.8 C-Code blocks. Legacy models that contain C-Code blocks cannot be ported using this tool.

5.3.8 Managing Block Diagram Global-Constant Values at Runtime

acslX provides a graphical dialog box for use in managing block diagram CONSTANT values at runtime. This dialog, however, cannot be used to manage GLOBAL CONSTANT values. The SET command must be used to change the value of GLOBAL CONSTANTS at runtime. See Chapter 4 of the acslX User's Guide for more information.

Chapter 6 Migrating Legacy Command Files (.cmd) and M-files (.m)

ACSL 11.8 supported two general mechanisms for scripting simulation runs and performing analysis of simulation data. ACSL Sim included a command prompt from which commands for controlling the execution of a simulation could be issued; the command set included commands for displaying and setting the values of simulation variables, scripting repeated runs, performing linear analysis, and primitive plotting facilities. ACSL Math also included a command prompt, from which a rich set of analysis and plotting commands could be executed using the ACSL Math language. Both command interpreters supported script files, so that commands could be put into files and run in batch mode as an alternative to interactively entering them.

In acslX however, there is only one prompt featuring the syntax and conventions of ACSL Math m-files with the addition of the required simulation execution commands (start, prepare, output etc) included for use at this new *combined* prompt (**>**).

ACSL Math m-files should be added directly to the acslX project. With a few exceptions, the ACSL Math command language is supported in acslX with minimal changes.

ACSL Sim Command (.CMD) files will need to be converted to acslX m-files in order to be compatible with the new *combined* run-time prompt. Nearly every element of the ACSL Sim command language has an m-language equivalent. When first converting your ACSL projects, acslX will convert your existing .CMD files by adding a leading and trailing !! to the file.

6.1 Simulation Runtime Commands

ACSL 11.8 Sim Runtime Commands are referred to as Simulation Control Commands in acslX. See the acslX Command Reference Manual for additional details.

6.1.1 Unsupported ACSL SIM Runtime Commands

The following simulation control commands are not currently supported in acslX. Any m-files which use one or more of the commands listed here will need to be modified to remove the unsupported command

ANALYZE	QUIT
EXIT	RANGE
PAUSE	

6.1.2 Loading and Unloading a Simulation

ACSL 11.8 Sim automatically loads the simulation executable when the simulation is run from within the ACSL Builder application. Likewise, acslX will automatically load and unload the simulation executable at the appropriate time (e.g., unload before build, load after build or when project is opened). Alternatively, a simulation may be manually loaded or unloaded using the new ***load*** and ***unload*** simulation control commands. See the acslX Command Reference Manual for more information.

6.1.3 Changes to the Use of Runtime Script Files

One significant difference between ACSL and acslX is the use of runtime scripts and files. In ACSL users could use either CMD files or m-files. In acslX, all runtime scripts must be converted to m-files. In ACSL 11.8, a CMD file with the name of the simulation DLL is searched for when the simulation is loaded; if one is found, it is executed. In acslX, no m-file file is executed unless one is placed in the project. Every m-file placed in the project will be executed when the simulation is loaded if the "*Execute at Load-Time*" option is selected from the m-files context menu.

6.1.4 Syntax Changes to Specific Commands

With the introduction of the new runtime prompt, certain commands, while still supported, have different syntax than in ACSL 11.8. These commands are listed below, along with a brief summary of the syntax changes. For a full description of the new syntax, refer to the acslX Command Reference.

The following list summarizes the syntax changes between ACSL 11.8 and acslX Simulation Control Commands.

- *Case sensitivity* – all commands and variable names used at the runtime prompt are case sensitive. By default, all intrinsic m-function names are lowercase, while the instantaneous values of simulation variables are all uppercase
- ***continue*** – All 11.8 flags have been depreciated. ***@nocallback*** and ***@[no]blocking*** flags have been added. See the description of "Synchronous and Non synchronous Execution" in the API section in the next chapter for details, or refer to the acslX Command Reference Manual.

- ***display*** - All 11.8 flags have been depreciated. ***@all @constants @variables*** flags have been added. See the Command Reference Manual for more details. The instantaneous values of simulated variables will be echoed when their name is typed at the (**>**) prompt in all uppercase.
- ***file*** – the command is identical to the m-file equivalent from ACSL Math. Please refer to the acslX Command Reference Manual for more details.
- ***output*** - All 11.8 flags have been depreciated. ***@all @clear @Nciout=NNN*** flags have been added. See the Command Reference Manual for more details. The instantaneous values of simulated variables will be echoed when their name is typed at the (**>**) prompt in all uppercase.
- ***plot*** – the command is identical to the m-file equivalent from Acsl Math. Please refer to the acslX Command Reference Manual for more details.
- ***set*** – This command has been depreciated. Model constants are now literally assigned values. (i.e. *VARNAME_INCALLCAPS = 10.0*)
- ***start*** - All ACSL 11.8 flags have been deprecated. ***@nocallback*** and ***@[no]blocking*** flags have been added. See the description of "Synchronous and Non synchronous Execution" in the API section in the next chapter for details, or refer to the acslX Command Reference.

6.2 ACSL Math Command/Functions and the acslX Analysis Language

ACSL Math has traditionally been sold as an add-on product to ACSL. And provides a plotting, analysis, and scripting capability based on the M-file language. With acslX, this analysis functionality is an integral element of the software and is referred to as the acslX analysis language. See the acslX Command Reference manual for details in the acslX Analysis Language.

Changes to supported analysis commands are generally restricted to commands pertaining to the generation of various plots. In particular, all commands related to low-level plot primitives have been removed. Specific command for the creation of a particular type of plot using passed arrays of data are still supported, but these commands no longer accept the previously supported optional parameters for specifying line color, line width, symbol type, etc.

Generally, the only data that should be passed to any of the supported commands for the creation of plot are the matrices containing the data to be plotted. Any modification of the plot settings should be done through the plot properties dialog within the acslX IDE. Note that the plot command now accepts a string specifying a file name containing plot settings; this argument may be passed to one of the supported plot command to force the plot to read settings (line colors, width, etc.) from the configuration file after loading the data to be plotted. The configuration files can be created by adjusting the settings from the properties dialog, then selecting the option to save these settings in the native format of the plot control. See the User's Guide for information on configuration of plots within acslX.

6.2.1 Unsupported ACSL Math Commands

The following analysis commands are not currently supported in acslX. Any M-file which uses one or more of the commands listed here will need to be modified to remove the unsupported command.

axes	drawnow	hold	Stem
axis	edit	image	Subplot
blt	feather	ishold	Surface
cinvert	figure	line	Text
cla	fill	llch	Uicontrol
clf	fillplot	newplot	View
close	gcf	patch	Viewmtx
colstyle	get	reset	Xlabel
cylinder	grid	rose	Ylabel
datalim	gtext	set	Zlabel
delete	hidden	sphere	

Generally, these commands correspond to low-level graphics primitives or advanced plot settings. Advanced plot configuration is now accomplished via the plot properties dialog within the acslX IDE. See previous chapters of the acslX User's Guide for more information on plotting within the acslX IDE.

6.3 Changes to Graphic Model Variable Naming Convention

Changes have been made to the manner in which target language variable names are constructed from variables contained in Graphic Model CSL/GSL code. An understanding of the changes to this mechanism is important when porting Cmd or M-files which make explicit use of model variables. The variable names automatically generated by acslX when legacy models are rebuilt will likely be different than those generated with ACSL 11.8.

In ACSL 11.8, block names only needed to be unique within the scope of their parent block in the hierarchy of the block diagram. To ensure that variable names were unique in the generated code, the default names were constructed by prepending the names of all parent blocks up the hierarchy of the block diagram, and appending instance numbers where necessary to ensure unique block names. For example, the initial condition variable contained in an integrator block contained in a compound block called "sensor" might produce a variable in the generated code called "sensor_sci2_3ic." This variable name could be used in an M-file to control the initial condition on this integrator. In complex or highly nested models, this could potentially produce very long variable names which violated name length constraints on some compilers. To remedy this, long variable names are replaced in ACSL 11.8 by dummy variables of the form Znnnnn.

In acslX, block name uniqueness is enforced globally, so it is only necessary to prepend the name of the block containing the CSL code to the variable name to guarantee uniqueness. An underscore is placed between the block and variable name for readability. In the above example, the variable name in the generated code would be "sci2_3_ic." In complex models, this results in much shorter variable names, so dummy replacement of long variable names is no longer necessary. Moreover, construction of analysis or simulation control commands which directly manipulate model variables is greatly simplified.

This change in variable name construction logic requires examination of any commands of M-files which are used to control execution or analysis of legacy block diagrams. As part of the porting process, care should be taken to replace explicit use of model variables using the old name construction logic with the newer short variable names. When unsure of the name of a particular variable, it is often helpful to display the values of all variables after building the simulation using the "**display @all**" command at the **(>)** prompt. This lists all model variables as they appear in the generated code. Block names are available by either examining the block properties, or hovering the mouse over the output port associated with a particular wire; this will display the name of the block and variable associated with the wire. Construction of the variable name is done by simply prepending the block name followed by an underscore to the variable name.

Chapter 7 Using the New acslX Simulation API

acslX introduces a number of significant modifications to the way in which simulations can be integrated into user applications using the simulation API. A primary objective of updating the simulation API has been to facilitate integration of acslX simulations with a wider range of programming languages, software environments, and communication/component technologies. In addition, a major goal of the new API was to pave the way for future acslX functionality.

The following sections summarize differences between the ACSL 11.8 and acslX simulation APIs. For a detailed description of acslX simulation API services, along with example usage, refer to Appendix B of the acslX User's Guide.

7.1 Similarities and Differences Between the New and Old API's

In general, the semantic nature of services comprising the acslX simulation API closely follows those of the ACSL 11.8 API. There are, for example, services for converting a variable name to an index in the model dictionary (and vice-versa). Similarly, services exist for getting or setting the value of a particular variable given its index in the dictionary.

The most important distinction between the ACSL 11.8 and acslX simulation APIs is the choice of the framework technology on which the respective APIs are based. ACSL 11.8 API is implemented as a COM (OLE) component, which makes it very straightforward to integrate an ACSL 11.8 simulation with environments like Visual Basic or Visual C++, but severely limits the ability to integrate ACSL 11.8 simulation with applications that do not support COM.

In contrast, the native acslX simulation API is a simple set of C-callable routines exported by the generated simulation executable. Built on top of this native API are software wrappers, or bindings, which adapt the native API for use in various environments or programming languages. At present, acslX also includes a .Net binding, which allows acslX simulations to be integrated with any of the various languages supported by the Microsoft .Net Framework: C#, Visual Basic .Net and managed C++ are among these.

One other major distinction between ACSL 11.8 and acslX simulations in general is the relative placement of command interpreter logic within the software architecture. In ACSL 11.8 and previous versions, the ACSL Sim command interpreter was effectively embedded into any generated simulation executable. Consequently, the ACSL 11.8 API includes a routine for sending a command string to the simulation for execution: HRESULT Command (const char *cmd). In acslX, all command interpretation is done externally to the simulation executable; as a result no API exists for programmatically passing command strings to the simulation for interpretation.

7.2 API Syntax

Appendix B of the acslX User's Guide provides syntax and example use of the acslX VB.NET API.

7.3 Use of Callback Routines

The mechanism by which information is passed from the simulation to a client application in acslX is based on callback functions. In contrast, ACSL 11.8 requires that the client implement a COM interface defined in the acslapi.h header file. Prototypes for the acslX callback functions are defined in AcslSimApi.h; client code must provide implementations of these functions and register them with the simulation using one of the AcslAddXXXCallback services.

7.4 Synchronous and Non-synchronous Simulation Execution

The ACSL 11.8 OPEN API, simulations were started using the Start() service, which blocked communications until simulation execution completed. Blocking communications is still supported in acslX via the AcslSimStartBlocking service, but the AcslSimStart service provides an alternative mechanism for starting simulation execution in a thread other than the calling thread. This allows the client application to resume processing while the simulation runs concurrently. Notifications of the simulation's state of execution may be monitored using the AcslExecutionStateChangeCallbackFunction; in particular, this is the mechanism by which the client is notified that simulation execution has completed. For legacy ACSL API applications in which blocking execution is desired, simply use the blocking form of the start service.

7.5 Debugging Routines

A number of debugging routines have been added to the API. These are used by the acslX integrated debugger, and they are also available for use in user-created client applications. These services include mechanisms for setting or clearing breakpoints, starting debug execution, stepping and stopping. See Appendix B of the acslX User's Guide for details on these services.

7.6 Routines and Arguments Reserved for Future Use

A number of routines have been included in the AcslSimAPI.h file for use in a future release of acslX, and are not yet implemented. These are documented in the AcslSimApi.h header file and in Appendix B of the acslX User's Guide. Currently, these services should not be used. Similarly, certain functions contain arguments which are reserved for future use; specifically, many of the services take an optional first argument "instanceHandle" which should always be set to a value of 1.

7.7 Language Bindings for the New API

Currently, acslX supports APIs which are callable from either C/C++ code, or from applications written using Microsoft's .Net platform. A COM interface is not supported in the current release of acslX.

7.8 Integration of acslX Simulation with C/C++ Code

Natively, acslX simulations support a simple C-callable API defined in the AcslSimAPI.h header file. Integration of user-supplied client code which is compatible with C-callable library routines may make use of these routines in a number of ways.

The most straightforward way for C/C++ language code to call acslX API routines is to use the optionally generated export library which is constructed when the simulation is built. Client code may simply include the AcslSimAPI.h file and link with this export library (the library will typically have the name of the simulation DLL).

Alternatively, client code may explicitly get the addresses of the required API routines by fetching a pointer to the routine using operating system services for extracting a pointer to an exported symbol from a dynamically linked library (e.g., in the Win32 API, the GetProcAddress() function could be used). See the documentation for your particular compiler and operating system API for further details.

7.9 Integration of acslX Simulations with .NET Applications (C#, VB.NET)

Use of the .Net binding is generally similar to the use of the ACSL 11.8 API COM API. All services defined in the AcslSimAPI.h header file have corresponding methods in the AcslSimulation class. Similarly, callback functions are represented by .Net interfaces. To use the .Net binding in your applications, the AcslSimulation.dll and AcslBaseTypes.dll assemblies must be included in your project. Once this is done, classes and interfaces defined in the assemblies may be used in your code in the same manner as other .Net components. See your compiler's documentation for details.