Licensee acknowledges that use of Fluent Inc.’s products can only provide an imprecise estimation of possible future performance and that additional testing and analysis, independent of the Licensor’s products, must be conducted before any product can be finally developed or commercially introduced. As a result, Licensee agrees that it will not rely upon the results of any usage of Fluent Inc.’s products in determining the final design, composition or structure of any product.

Copyright © 2003 by Fluent Inc.
All rights reserved. No part of this document may be reproduced or otherwise used in any form without express written permission from Fluent Inc.

Airpak, FIDAP, FLUENT, GAMBIT, Icepak, MixSim, and POLYFLOW are registered trademarks of Fluent Inc. All other products or name brands are trademarks of their respective holders.

Fluent Inc.
Centerra Resource Park
10 Cavendish Court
Lebanon, NH 03766
About This Document

User-defined functions (UDFs) allow you to customize FLUENT and can significantly enhance its capabilities. This UDF Manual presents detailed information on how to write, compile, and use UDFs in FLUENT. Examples have also been included, where available.

Information in this manual is presented in the following chapters:

- Chapter 1: Overview
- Chapter 2: C Programming Basics for UDFs
- Chapter 3: A Step-by-Step UDF Example
- Chapter 4: DEFINE Macros
- Chapter 5: Accessing Solver Data
- Chapter 6: Utilities
- Chapter 7: Interpreting and Compiling Source Files
- Chapter 8: Hooking a UDF to FLUENT
- Chapter 9: Parallel UDF Usage
- Chapter 10: User-Defined Scalar (UDS) Transport Modeling
- Chapter 11: Sample Problems

This document provides some basic information about the C programming language as it relates to UDFs in FLUENT and assumes that you have some programming experience with C. If you are unfamiliar with C, please consult a C language reference (e.g., [2]) for basic information.

This document does not imply responsibility on the part of Fluent Inc. for the accuracy or stability of solutions obtained using UDFs that are either user-generated or provided by Fluent Inc. Support for current license holders will be limited to guidance related to communication between a UDF and the FLUENT solver. Other aspects of the UDF development process that include conceptual function design, implementation (writing C code), compilation and debugging of C source code, execution of the UDF, and function design verification will remain the responsibility of the UDF author.
## Contents

1 Overview 1-1

1.1 What is a User-Defined Function (UDF)? 1-1
1.2 Why Use UDFs? 1-3
1.3 Limitations 1-3
1.4 Defining Your UDF Using DEFINE Macros 1-4
1.5 Including the udf.h Header File in Your Source File 1-5
1.6 Differences Between Interpreted and Compiled UDFs 1-6
1.7 Grid Terminology 1-8
1.8 Data Types in FLUENT 1-10
1.8.1 Multiphase-specific Data Types 1-11
1.9 Calling Sequence of a UDF in the FLUENT Solution Process 1-13
1.10 Special Considerations for Multiphase UDFs 1-16

2 C Programming Basics for UDFs 2-1

2.1 Introduction 2-1
2.2 Commenting Your C Code 2-2
2.3 C Data Types in FLUENT 2-2
2.4 Constants 2-2
2.5 Variables 2-3
2.5.1 Declaring Variables 2-4
2.5.2 External Variables 2-4
2.5.3 Static Variables 2-6
2.6 User-Defined Data Types 2-7
2.7 Casting 2-7
2.8 Functions 2-7
2.9 Arrays ....................................................... 2-8
2.10 Pointers .................................................. 2-8
2.11 Control Statements ...................................... 2-10
   2.11.1 if Statement .................................... 2-10
   2.11.2 if-else Statement ............................... 2-10
   2.11.3 for Loops ....................................... 2-11
2.12 Common C Operators ................................... 2-12
   2.12.1 Arithmetic Operators .......................... 2-12
   2.12.2 Logical Operators .............................. 2-12
2.13 C Library Functions .................................... 2-13
   2.13.1 Trigonometric Functions ....................... 2-13
   2.13.2 Miscellaneous Mathematical Functions ........ 2-13
   2.13.3 Standard I/O Functions ....................... 2-14
2.14 Preprocessor Directives ................................ 2-16
2.15 Comparison with FORTRAN ........................... 2-17

3 A Step-by-Step UDF Example 3-1

3.1 Process Overview ....................................... 3-1
   3.1.1 Step 1: Define Your Problem ................... 3-2
   3.1.2 Step 2: Create a C Source File ................. 3-4
   3.1.3 Step 3: Start FLUENT and Read (or Set Up) the Case File 3-5
   3.1.4 Step 4: Interpret the Source File .............. 3-5
   3.1.5 Step 5: Hook Your UDF to FLUENT ............. 3-7
   3.1.6 Step 6: Run the Calculation .................... 3-7
   3.1.7 Step 7: Analyze the Numerical Solution and Compare to Expected Results 3-8
4 DEFINE Macros

4.1 Introduction ................................................. 4-1

4.2 General DEFINE Macros ................................. 4-2
  4.2.1 DEFINE_ADJUST .................................. 4-3
  4.2.2 DEFINE_EXECUTE_AT_END ......................... 4-6
  4.2.3 DEFINE_INIT ...................................... 4-8
  4.2.4 DEFINE_ON_DEMAND ................................. 4-10
  4.2.5 DEFINE_RW_FILE .................................. 4-13

4.3 Model-Specific DEFINE Macros ......................... 4-15
  4.3.1 DEFINE_CHEM_STEP ................................ 4-19
  4.3.2 DEFINE_DELTAT ..................................... 4-21
  4.3.3 DEFINE_DIFFUSIVITY ............................... 4-23
  4.3.4 DEFINE_DOM_DIFFUSE_REFLECTIVITY ............... 4-25
  4.3.5 DEFINE_DOM_SOURCE ................................ 4-27
  4.3.6 DEFINE_DOM_SPECULAR_REFLECTIVITY .............. 4-29
  4.3.7 DEFINE_HEAT_FLUX .................................. 4-32
  4.3.8 DEFINE_NET_REACTION_RATE ...................... 4-34
  4.3.9 DEFINE_NOX_RATE ................................... 4-36
  4.3.10 DEFINE_PR_RATE .................................... 4-40
  4.3.11 DEFINE_PRANDTL .................................... 4-46
  4.3.12 DEFINE_PROFILE .................................... 4-53
  4.3.13 DEFINE_PROPERTY ................................... 4-63
  4.3.14 DEFINE_SCAT_PHASE_FUNC ......................... 4-69
  4.3.15 DEFINE_SOURCE .................................... 4-72
  4.3.16 DEFINE_SR_RATE .................................... 4-76
  4.3.17 DEFINE_TURB_PREMIX_SOURCE ...................... 4-82
  4.3.18 DEFINE_TURBULENT_VISCOSITY ..................... 4-85
  4.3.19 DEFINE_UDS_FLUX ................................... 4-87
CONTENTS

4.3.20 DEFINE_UDS_UNSTEADY ............................................. 4-91
4.3.21 DEFINE_VR_RATE ..................................................... 4-93
4.4 Multiphase DEFINE Macros ........................................... 4-97
  4.4.1 DEFINE_CAVITATION_RATE ....................................... 4-98
  4.4.2 DEFINE_EXCHANGE_PROPERTY .................................... 4-100
  4.4.3 DEFINE_VECTOR_EXCHANGE_PROPERTY ........................... 4-105
4.5 Dynamic Mesh DEFINE Macros ...................................... 4-107
  4.5.1 DEFINE.CG.MOTION .................................................. 4-108
  4.5.2 DEFINE_GEOM ........................................................ 4-111
  4.5.3 DEFINE_GRID.MOTION .............................................. 4-113
4.6 DPM DEFINE Macros ................................................. 4-116
  4.6.1 DEFINE_DPM_BC ...................................................... 4-118
  4.6.2 DEFINE_DPM_BODY_FORCE ....................................... 4-126
  4.6.3 DEFINE_DPM_DRAG .................................................. 4-128
  4.6.4 DEFINE_DPM_EROSION .............................................. 4-130
  4.6.5 DEFINE_DPM_INJECTION_INIT .................................... 4-136
  4.6.6 DEFINE_DPM_LAW .................................................... 4-139
  4.6.7 DEFINE_DPM_OUTPUT ............................................... 4-141
  4.6.8 DEFINE_DPMPROPERTY ............................................. 4-143
  4.6.9 DEFINE_DPM_SCALAR_UPDATE .................................... 4-147
  4.6.10 DEFINE_DPM_SOURCE ............................................. 4-151
  4.6.11 DEFINE_DPM_SPRAY_COLLIDE ................................... 4-153
  4.6.12 DEFINE_DPM_SWITCH .............................................. 4-156

5 Accessing Solver Data ................................................. 5-1
  5.1 Introduction .......................................................... 5-1
  5.2 Grid Variables ....................................................... 5-3
    5.2.1 Node Position Variables ....................................... 5-3
    5.2.2 Connectivity Variables ........................................ 5-3
5.3 Cell Variables ........................................... 5-9
5.4 Face Variables ......................................... 5-21
5.5 Multiphase Variables ................................... 5-23
5.6 DPM Variables ......................................... 5-24
5.7 NOx Variables ......................................... 5-28

6 Utilities ..................................................... 6-1
6.1 Introduction ............................................. 6-1
6.2 Looping Macros ......................................... 6-2
6.3 Multiphase-Specific Looping Macros ................. 6-6
6.4 Setting Face Variables Using F_PROFILE ........... 6-10
6.5 Accessing Variables That Are Not Passed as Arguments .... 6-11
6.6 Vector Macros .......................................... 6-18
6.7 Time-Dependent Macros ............................... 6-22
6.8 User-Defined Scheme Variables ..................... 6-24
6.9 Input/Output Macros ................................... 6-26
6.10 Additional Macros ..................................... 6-27

7 Interpreting and Compiling Source Files ................. 7-1
7.1 Introduction ............................................. 7-1
7.1.1 Interpreted UDFs .................................... 7-1
7.1.2 Compiled UDFs ...................................... 7-2
7.1.3 Location of the udf.h File ......................... 7-2
7.2 Interpreting a Source File Using the Interpreted UDFs Panel .... 7-3
7.2.1 Common Errors Made While Interpreting Your Source File ... 7-6
7.3 Compiling a Source File Using the Compiled UDFs Panel ........ 7-7
7.3.1 Common Errors Made While Loading Your Compiled Library ... 7-11
8 Hooking Your UDF to FLUENT 8-1

8.1 Hooking General Solver UDFs to FLUENT 8-2

8.1.1 DEFINE_ADJUST 8-2
8.1.2 DEFINE_EXECUTE_AT_END 8-3
8.1.3 DEFINE_INIT 8-4
8.1.4 DEFINE_ON_DEMAND 8-5
8.1.5 DEFINE_RW_FILE 8-6
8.1.6 User-Defined Memory Storage 8-7

8.2 Hooking Model-Specific UDFs to FLUENT 8-8

8.2.1 DEFINE_CHEM_STEP 8-8
8.2.2 DEFINE_DELTAT 8-9
8.2.3 DEFINE_DIFFUSIVITY 8-10
8.2.4 DEFINE_DOM_DIFFUSE_REFLECTIVITY 8-12
8.2.5 DEFINE_DOM_SOURCE 8-13
8.2.6 DEFINE_DOM_SPECULAR_REFLECTIVITY 8-14
8.2.7 DEFINE_HEAT_FLUX 8-15
8.2.8 DEFINE_NET_REACTION_RATE 8-16
8.2.9 DEFINE_NOX_RATE 8-17
8.2.10 DEFINE_PR_RATE 8-18
8.2.11 DEFINE_PRANDTL 8-19
8.2.12 DEFINEPROFILE 8-20
8.2.13 DEFINEPROPERTY 8-21
8.2.14 DEFINE_SCAT_PHASE_FUNC 8-23
8.2.15 DEFINE_SOURCE 8-25
8.2.16 DEFINE_SR_RATE 8-26
8.2.17 DEFINE_TURB_PREMIX_SOURCE 8-27
8.2.18 DEFINE_TURBULENT_VISCOSITY 8-28
8.2.19 DEFINE_UDS_FLUX 8-29
8.2.20 DEFINE_UDS_UNSTEADY ...................................................... 8-30
8.2.21 DEFINE_VR_RATE ............................................................. 8-31
8.3 Hooking Multiphase UDFs to FLUENT ................................. 8-32
  8.3.1 DEFINE_CAVITATION_RATE ........................................... 8-32
  8.3.2 DEFINE_EXCHANGE_PROPERTY ....................................... 8-34
  8.3.3 DEFINE_VECTOR_EXCHANGE_PROPERTY ............................ 8-36
8.4 Hooking Dynamic Mesh UDFs to FLUENT .............................. 8-38
  8.4.1 DEFINE.CG_MOTION .................................................... 8-38
  8.4.2 DEFINE_GEOM .......................................................... 8-40
  8.4.3 DEFINE_GRID_MOTION ............................................... 8-42
8.5 Hooking DPM UDFs to FLUENT ........................................... 8-44
  8.5.1 DEFINE_DPM_BC ..................................................... 8-44
  8.5.2 DEFINE_DPM_BODY_FORCE ........................................ 8-46
  8.5.3 DEFINE_DPM_DRAG .................................................. 8-47
  8.5.4 DEFINE_DPM_EROSION .............................................. 8-48
  8.5.5 DEFINE_DPM_INJECTION_INIT .................................... 8-50
  8.5.6 DEFINE_DPM_LAW .................................................... 8-51
  8.5.7 DEFINE_DPM_OUTPUT ............................................... 8-52
  8.5.8 DEFINE_DPM_PROPERTY ............................................. 8-53
  8.5.9 DEFINE_DPM_SCALAR_UPDATE .................................... 8-55
  8.5.10 DEFINE_DPM_SOURCE ............................................. 8-56
  8.5.11 DEFINE_DPM_SPRAY_COLLIDE .................................... 8-57
  8.5.12 DEFINE_DPM_SWITCH ............................................ 8-58

9 Parallel UDF Usage ............................................................ 9-1
  9.1 Overview of Parallel FLUENT ......................................... 9-1
    9.1.1 Command Transfer and Communication .......................... 9-4
  9.2 Cells and Faces in a Partitioned Grid ............................. 9-7
  9.3 Parallelizing Your Serial UDF ....................................... 9-11
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.4</td>
<td>Macros for Parallel UDFs</td>
<td>9-12</td>
</tr>
<tr>
<td>9.4.1</td>
<td>Compiler Directives</td>
<td>9-12</td>
</tr>
<tr>
<td>9.4.2</td>
<td>Communicating Between the Host and Node Processes</td>
<td>9-15</td>
</tr>
<tr>
<td>9.4.3</td>
<td>Predicates</td>
<td>9-17</td>
</tr>
<tr>
<td>9.4.4</td>
<td>Global Reduction Macros</td>
<td>9-18</td>
</tr>
<tr>
<td>9.4.5</td>
<td>Looping Macros</td>
<td>9-22</td>
</tr>
<tr>
<td>9.4.6</td>
<td>Cell and Face Partition ID Macros</td>
<td>9-28</td>
</tr>
<tr>
<td>9.4.7</td>
<td>Message Displaying Macros</td>
<td>9-29</td>
</tr>
<tr>
<td>9.4.8</td>
<td>Message Passing Macros</td>
<td>9-30</td>
</tr>
<tr>
<td>9.4.9</td>
<td>Macros for Exchanging Data Between Compute Nodes</td>
<td>9-34</td>
</tr>
<tr>
<td>9.5</td>
<td>Process Identification</td>
<td>9-35</td>
</tr>
<tr>
<td>9.6</td>
<td>Parallel UDF Example</td>
<td>9-36</td>
</tr>
<tr>
<td>9.7</td>
<td>Writing Files in Parallel</td>
<td>9-39</td>
</tr>
<tr>
<td>10</td>
<td>User-Defined Scalar (UDS) Transport Modeling</td>
<td>10-1</td>
</tr>
<tr>
<td>10.1</td>
<td>Introduction</td>
<td>10-1</td>
</tr>
<tr>
<td>10.2</td>
<td>Theory</td>
<td>10-1</td>
</tr>
<tr>
<td>10.3</td>
<td>Defining, Solving, and Postprocessing a UDS</td>
<td>10-3</td>
</tr>
<tr>
<td>11</td>
<td>Sample Problems</td>
<td>11-1</td>
</tr>
<tr>
<td>11.1</td>
<td>Boundary Conditions</td>
<td>11-1</td>
</tr>
<tr>
<td>11.2</td>
<td>Source Terms</td>
<td>11-12</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Adding a Momentum Source to a Duct Flow</td>
<td>11-12</td>
</tr>
<tr>
<td>11.3</td>
<td>Physical Properties</td>
<td>11-18</td>
</tr>
<tr>
<td>11.3.1</td>
<td>Solidification via a Temperature-Dependent Viscosity</td>
<td>11-18</td>
</tr>
<tr>
<td>11.4</td>
<td>Reaction Rates</td>
<td>11-23</td>
</tr>
<tr>
<td>11.4.1</td>
<td>A Custom Volume Reaction Rate</td>
<td>11-23</td>
</tr>
<tr>
<td>11.5</td>
<td>User-Defined Scalars</td>
<td>11-28</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Postprocessing Using User-Defined Scalars</td>
<td>11-28</td>
</tr>
<tr>
<td>11.5.2</td>
<td>Implementing FLUENT’s P-1 Radiation Model</td>
<td>11-31</td>
</tr>
</tbody>
</table>
A DEFINE Macro Definitions
A.1 General Solver DEFINE Macros ......................................... A-1
A.2 Model-Specific DEFINE Macros ........................................ A-2
A.3 Multiphase DEFINE Macros ............................................. A-4
A.4 Dynamic Mesh Model DEFINE Macros ............................... A-5
A.5 Discrete Phase Model DEFINE Macros .............................. A-6
Chapter 1. Overview

This chapter contains an overview of user-defined functions (UDFs) and their usage in FLUENT. Details about UDF functionality are described in the following sections:

- Section 1.1: What is a User-Defined Function (UDF)?
- Section 1.2: Why Use UDFs?
- Section 1.3: Limitations
- Section 1.4: Defining Your UDF Using DEFINE Macros
- Section 1.5: Including the udf.h Header File in Your UDF Source File
- Section 1.6: Differences Between Interpreted and Compiled UDFs
- Section 1.7: Grid Terminology
- Section 1.8: Data Types in FLUENT
- Section 1.9: Calling Sequence for a UDF in the Solution Process
- Section 1.10: Special Considerations for Multiphase UDFs

1.1 What is a User-Defined Function (UDF)?

A user-defined function, or UDF, is a function that you program that can be dynamically loaded with the FLUENT solver to enhance the standard features of the code. UDFs are written in the C programming language. They are defined using DEFINE macros that are supplied by Fluent Inc. They access data from the FLUENT solver using predefined macros and functions also supplied by Fluent Inc. Every UDF contains the udf.h file inclusion directive (#include "udf.h") at the beginning of the source code file, which allows definitions for DEFINE macros and other Fluent-provided macros and functions to be included during the compilation process. UDFs are executed as either interpreted or compiled functions in FLUENT. Values that are passed to the solver by a UDF or returned by the solver to a UDF must be specified in SI units.
In summary, UDFs:

- are written in the C programming language. (Chapter 2)
- must have an include statement for the udf.h file. (Section 1.5)
- must be defined using DEFINE macros supplied by Fluent Inc. (Section 1.4)
- access FLUENT solver data using predefined macros and functions supplied by Fluent Inc. (Chapters 5 and 6)
- are executed as interpreted or compiled functions. (Chapter 7)
- must have all values returned to the FLUENT solver specified in SI units.

User-defined functions can perform a variety of tasks in FLUENT. They can return a value unless they are defined as void in the udf.h file. If they do not return a value, they can modify an argument, modify a variable not passed as an argument, or perform I/O tasks with case and data files. In summary, UDFs can:

- return a value.
- modify an argument.
- return a value and modify an argument.
- modify a FLUENT variable (not passed as an argument).
- write information to (or read information from) a case or data file.

UDFs are written in C using any text editor and the source file is saved with a .c file extension. Source files typically contain a single UDF, but they can contain multiple, concatenated functions. Source files can be either interpreted or compiled in FLUENT. For interpreted UDFs, source files (e.g., pressure_profile.c) are interpreted and loaded directly at runtime, in a single-step process. For compiled UDFs, the process involves two separate steps. A shared object code library is first built and then it is loaded into FLUENT. Once interpreted or compiled, UDFs will become visible and selectable in FLUENT graphics panels, and can be hooked to a solver by choosing the function name in the appropriate panel (see Chapter 8).
1.2 Why Use UDFs?

UDFs allow you to customize FLUENT to fit your particular modeling needs. UDFs can be used for a variety of applications, some of which are listed below:

- Customization of boundary conditions, material property definitions, surface and volume reaction rates, source terms in FLUENT transport equations, source terms in user-defined scalar (UDS) transport equations, diffusivity functions, etc.
- Adjustment of computed values on a once-per-iteration basis.
- Initialization of a solution.
- Asynchronous execution of a UDF (on demand).
- Post-processing enhancement.
- Enhancement of existing FLUENT models (e.g., discrete phase model, multiphase mixture model, discrete ordinates radiation model).

Some examples of sample problems involving UDFs can be found in Chapters 10 and 11.

1.3 Limitations

Although the UDF capability in FLUENT can address a wide range of applications, it is not possible to address every application using UDFs. Not all solution variables or FLUENT models can be accessed by UDFs. Specific heat values, for example, cannot be modified; this would require additional solver capabilities. If you are unsure whether a particular problem can be handled using a UDF, you can contact your technical support engineer for assistance.
1.4 Defining Your UDF Using DEFINE Macros

UDFs are defined using Fluent-supplied function declarations. These function declarations are implemented in the code as macros, and are referred to in this document as DEFINE (all capitals) macros. Definitions for DEFINE macros are contained in the udf.h header file (see Appendix A for a listing). For a complete description of each DEFINE macro and an example of its usage, refer to Chapter 4.

The general format of a DEFINE macro is

\[
\text{DEFINE_MACRONAME(udf\_name, passed-in variables)}
\]

where the first argument in the parentheses is the name of your UDF. Name arguments are case-sensitive and must be specified in lowercase. The name that you choose for your UDF will become visible and selectable in drop-down lists in FLUENT, once the function has been interpreted or compiled. The second set of input arguments to the DEFINE macro are variables that are passed into your function from the FLUENT solver.

For example, the macro

\[
\text{DEFINE\_PROFILE(inlet\_x\_velocity, thread, index)}
\]

defines a profile function named inlet_x_velocity with two variables, thread and index, that are passed into the function from FLUENT. These passed-in variables are the boundary condition zone ID (as a pointer to the thread) and the index identifying the variable that is to be stored. Once the UDF has been compiled, its name (e.g., inlet_x_velocity) will become visible and selectable in drop-down lists in the appropriate boundary condition panel (e.g., Velocity Inlet panel) in FLUENT.

! Note that all of the arguments to a DEFINE macro need to be placed on the same line in your source code. Splitting the DEFINE statement onto several lines will result in a compilation error.
1.5 Including the udf.h Header File in Your Source File

The udf.h header file contains definitions for DEFINE macros as well as #include compiler directives for C library function header files. It also includes header files for other Fluent-supplied macros and functions (e.g., mem.h). You must, therefore, include the udf.h file at the beginning of every UDF source code file using the #include compiler directive:

```
#include "udf.h"
```

For example, when udf.h is included in the source file containing the DEFINE statement from the previous section,

```
DEFINE_PROFILE(inlet_x_velocity, thread, index)
```

upon compilation, the macro will expand to

```
void inlet_x_velocity(Thread *thread, int index)
```

You won’t need to put a copy of udf.h in your local directory when you compile your UDF. The FLUENT solver automatically reads the udf.h file from the Fluent.Inc/fluent6.x/src/ directory once your UDF is compiled.

It is customary practice in C programming to place all function prototypes in a header file. Therefore, if your UDF utilizes custom functions, then additional custom header files may need to be included in udf.h as well.
1.6 Differences Between Interpreted and Compiled UDFs

Compiled UDFs

Compiled UDFs are built in the same way that the FLUENT executable itself is built. A script called Makefile is used to invoke the system C compiler to build an object code library. (The object code library contains the native machine language translation of your higher-level C source code.) This shared library is then loaded into FLUENT at runtime by a process called “dynamic loading.” The object libraries are specific to the computer architecture being used, as well as to the particular version of the FLUENT executable being run. The libraries must, therefore, be rebuilt any time FLUENT is upgraded, when the computer’s operating system level changes, or when the job is run on a different type of computer.

Compiled UDF(s) are compiled from source code using the graphical user interface, in a two-step process. The process involves a visit to the Compiled UDFs panel where you first Build shared library object file(s) from a source file, and then Load the shared library into FLUENT.

Interpreted UDFs

Interpreted UDFs are also interpreted from source code using the graphical user interface, but in a single-step process. The process, which occurs at runtime, involves a visit to the Interpreted UDFs panel where you Interpret function(s) in a source file.

Inside FLUENT, the source code is compiled into an intermediate, architecture-independent machine code using a C preprocessor. This machine code then executes on an internal emulator, or interpreter, when the UDF is invoked. This extra layer of code incurs a performance penalty, but allows an interpreted UDF to be shared effortlessly between different architectures, operating systems, and FLUENT versions. If execution speed does become an issue, an interpreted UDF can always be run in compiled mode without modification.

The interpreter that is used for interpreted UDFs does not have all of the capabilities of a standard C compiler (which is used for compiled UDFs). Specifically interpreted UDFs cannot contain any of the following C programming language elements:

- goto statements
- non ANSI-C prototypes for syntax
- direct data structure references
- declarations of local structures
- unions
1.6 Differences Between Interpreted and Compiled UDFs

- pointers to functions
- arrays of functions

This major difference between interpreted and compiled UDFs can be significant if, for example, you want to introduce new data structures in your UDF. Interpreted cannot access FLUENT solver data using direct structure references; they can only indirectly access data through the use of Fluent-supplied macros. See Chapter 7 for details on compiled and interpreted UDFs in FLUENT.

Summary

When choosing between writing an interpreted UDF or a compiled UDF, keep in mind the following:

- **Interpreted UDFs:**
  - are portable to other platforms.
  - can all be run as compiled UDFs.
  - do not require a C compiler.
  - are slower than compiled UDFs.
  - are restricted in the use of the C programming language.
  - cannot be linked to compiled system or user libraries.
  - can only access data stored in a FLUENT structure using a predefined macro (see Chapters 5 and 6).

- **Compiled UDFs:**
  - execute faster than interpreted UDFs.
  - are not restricted in the use of the C programming language.
  - can be compiled with any ANSI-compliant C compiler.
  - can call functions written in other languages (specifics are system- and compiler-dependent).
  - cannot necessarily be run as interpreted UDFs if they contain certain elements of the C language that the interpreter cannot handle.
In summary, when deciding which type of UDF to use for your FLUENT model:

- use interpreted UDFs for small, straightforward functions.
- use compiled UDFs for complex functions that
  - have a significant CPU requirement (e.g., a property UDF that is called on a per-cell basis every iteration).
  - require access to a shared library.

1.7 Grid Terminology

Most user-defined functions access data from the FLUENT solver. Since solver data is defined in terms of grid components, you will need to learn some basic grid terminology before you can write a UDF.

A mesh is broken up into control volumes, or cells. Each cell is defined by a set of grid points (or nodes), a cell center, and the faces that bound the cell (Figure 1.7.1). FLUENT uses internal data structures to define the domain(s) of the mesh, assign an order to cells, cell faces, and grid points in a mesh, and establish connectivity between adjacent cells. A thread is the internal name of the data structure in FLUENT that is used to represent a (boundary or cell) zone. Cell threads are groupings of cells, and face threads are groupings of faces. A domain is the internal name of the data structure in FLUENT that is used to represent a grouping of node, face, and cell threads in a mesh.

Cells and cell faces are grouped into zones that typically define the physical components of the model (e.g., inlets, outlets, walls, fluid regions). A face will bound either one or two cells (identified by \( c_0 \) and \( c_1 \)) depending on whether it is a boundary face or an interior face. The cells on either side of a face may or may not belong to the same cell thread. If a face is on the boundary of a domain, then only \( c_0 \) exists. (\( c_1 \) is undefined for an external face). If the face is in the interior of the domain, then both \( c_0 \) and \( c_1 \) exist.
1.7 Grid Terminology

- **cell**: control volume into which domain is broken up
- **cell center**: location where cell data is stored
- **face**: boundary of a cell (2D or 3D)
- **edge**: boundary of a face (3D)
- **node**: grid point
- **cell thread**: grouping of cells
- **face thread**: grouping of faces
- **node thread**: grouping of nodes
- **domain**: a grouping of node, face, and cell threads

Figure 1.7.1: Grid Components
1.8 Data Types in FLUENT

In addition to standard C language data types such as `real`, `int`, etc. that can be used to define data in your UDF, there are FLUENT-specific data types that are associated with solver data. These data types represent the computational units for a grid (Figure 1.7.1). Variables that are defined using these data types are typically supplied as arguments to DEFINE macros as well as to other special functions that access FLUENT solver data.

Some of the more commonly-used FLUENT data types are:

- `cell_t`
- `face_t`
- `Thread`
- `Domain`
- `Node`

`cell_t` is the data type for a cell ID. It is an integer index that identifies a particular cell within a given thread.

`face_t` is the data type for a face ID. It is an integer index that identifies a particular face within a given thread.

The `Thread` data type is a structure that acts as a container for data that is common to the group of cells or faces that it represents. For multiphase applications, there is a thread structure for each phase, as well as for the mixture. See Section 1.8.1 for details.

The `Node` data type is a structure that acts as a container for data associated with the corner of a cell or face.

The `Domain` data type is a structure that acts as a container for data associated with a collection of node, face, and cell threads in a mesh. For single-phase applications, there is only a single domain structure. For multiphase applications, there are domain structures for each phase, the interaction between phases, as well as for the mixture. The mixture-level domain is the highest-level structure for a multiphase model. See Section 1.8.1 for details.

! Note that all of the FLUENT data types are case-sensitive.

When you use a UDF in FLUENT, your function can access solution variables at individual cells or cell faces in the fluid and boundary zones. UDFs need to be passed appropriate arguments such as a thread reference (i.e., pointer to a particular thread) and the cell or face ID in order to allow individual cells or faces to be accessed. Note that a face ID or cell ID, alone, does not uniquely identify the face or cell. A thread pointer is always required along with the ID to identify which thread the face (or cell) belongs to.

Some UDFs are passed the cell index variable (`c`) as an argument such as in `DEFINE_PROPERTY(my_function,c,t)`, or the face index variable (`f`) such as in `DEFINE_UDS_FLUX(my_function,f,t,i)`. If the cell or face index variable isn’t passed
as an argument, the variable is always available to be used in a UDF once it has been declared.

The data structures that are passed to your UDF (as pointers) depend on the DEFINE macro you are using and the property or term you are trying to modify. For example, DEFINE_ADJUST UDFs are general-purpose functions that are passed a domain pointer (d) such as in DEFINE_ADJUST(my_function, d). DEFINE_PROFILE UDFs are passed a thread pointer (t) to the boundary zone that the function is hooked to, such as in DEFINE_PROFILE(my_function, thread, i).

Some UDFs, such as DEFINE_ON_DEMAND functions, aren’t passed any pointers to data structures (e.g., DEFINE_ON_DEMAND(my_function)). If your UDF needs to access a thread or domain pointer that is not directly passed from the solver through an argument, then you will need to use a special Fluent-supplied utility (e.g., Get_Domain) to retrieve it. See Chapter 6 for details.

### 1.8.1 Multiphase-specific Data Types

When one of the multiphase models (i.e., Mixture, VOF, Eulerian) is enabled in FLUENT, properties and variables are stored for the mixture of all the phases, as well as for each individual phase. This additional storage for multiphase flows is accommodated in the code through the use of special thread and domain data structures.

In a multiphase application, the top-level domain is referred to as the ‘superdomain’. Each phase occupies a domain referred to as a ‘subdomain’. A third domain type, the ‘interaction’ domain, is introduced to allow for the definition of phase interaction mechanisms. When mixture properties and variables are needed (a sum over phases), the superdomain is used for those quantities while the subdomain carries the information for individual phases. In single-phase, the concept of a mixture is used to represent the sum over all the species (components) while in multiphase it represents the sum over all the phases. This distinction is important since the code will later be extended to multiphase multi-components (where, for example, a phase could be a mixture of species).

Since solver information is stored in thread data structures, threads must be associated with the superdomain as well as with each of the subdomains. That is, for each cell or face thread that is defined in the superdomain, there is a corresponding cell or face thread defined for each subdomain. Some of the information defined in one thread of the superdomain is shared with the corresponding threads of each of the subdomains. Threads associated with the superdomain are referred to as ‘superthreads’, while threads associated with the subdomain are referred to as phase-level threads, or ‘subthreads’. The domain and thread hierarchy are summarized in Figure 1.8.1.
Overview

Figure 1.8.1: Domain and Thread Structure Hierarchy

Figure 1.8.1 introduces the concept of the `domain_id` and `phase_domain_index`. The `domain_id` can be used in UDFs to distinguish the superdomain from the primary and secondary phase-level domains. The superdomain (mixture domain) `domain_id` is always assigned the value of 1. Interaction domains are also identified with the `domain_id`. The `domain_ids` are not necessarily ordered sequentially as shown in Figure 1.8.1.

The `phase_domain_index` can be used in UDFs to distinguish the primary phase-level threads from the secondary phase-level threads. For the primary phase-level thread, the `phase_domain_index` is always assigned the value of 0.
1.9 Calling Sequence of a UDF in the FLUENT Solution Process

UDFs are called at predetermined times in the FLUENT solution process. However, they can also be executed asynchronously (or “on demand”) using a `DEFINE_ON_DEMAND` function. If a `DEFINE_EXECUTE_AT_END` UDF is utilized, then FLUENT calls the function at the end of a timestep. Understanding the context in which UDFs are called within FLUENT’s solution process may be important when you begin the process of writing UDF code, depending on the type of UDF you are writing. The solver contains call-outs that are linked to user-defined functions that you write. Knowing the sequencing of function calls within an iteration in the FLUENT solution process can help you determine which data are current and available at any given time.

Segregated Solver

In the solution process for the segregated solver (Figure 1.9.1), a user-defined initialization function (defined using `DEFINE_INIT`) is executed before the iteration loop begins when you initialize the flow from the graphical user interface or text user interface. (Note that FLUENT calls the default initialization function first, and then calls the UDF.) The iteration loop then begins with the execution of user-defined adjust functions (defined using `DEFINE_ADJUST`). Following that, conservation equations are solved, progressing from the momentum equations and subsequent pressure correction equation to the additional scalar equations that are relevant to a particular calculation. Both the source and profile UDFs (defined using `DEFINE_SOURCE` and `DEFINE_PROFILE`, respectively) are called by each “Solve” routine for the variable currently under consideration (e.g., species, velocity).

After the conservation equations, the properties are updated (including user-defined properties). Thus, if your model involves the gas law, for example, the density will be updated at this time using the updated temperature (and pressure and/or species mass fractions). A check for either convergence or additional requested iterations is done, and the loop either continues or stops.
Overview

Figure 1.9.1: Solution Procedure for the Segregated Solver
Coupled Solver

In the solution process for the coupled solver (Figure 1.9.2), user-defined initialization functions (defined using `DEFINE_INIT`) are executed before the iteration loop begins when the solution is initialized. The iteration loop then begins with the execution of user-defined adjust functions (defined using `DEFINE_ADJUST`). Next, FLUENT solves the governing equations of continuity, momentum, and (where appropriate) energy simultaneously as a set, or vector, of equations. The rest of the solution procedure is the same as the segregated solver (Figure 1.9.1).

Figure 1.9.2: Solution Procedure for the Coupled Solver
1.10 Special Considerations for Multiphase UDFs

In many cases, the UDF source code that you will write for a single-phase flow will be the same as for a multiphase flow. For example, there will be no differences between the C code written for a single-phase boundary profile (defined using `DEFINE_PROFILE`) and the code for a multiphase profile, assuming that the function is only accessing data from the phase-level domain it is hooked to in the graphical user interface.

The data structures that are passed to a UDF depend on: 1) the multiphase model that is enabled, 2) which property or term is being modified, and 3) the `DEFINE` macro that is used, as well as the domain that is to be affected (mixture or phase). To better understand this, consider the differences between the Mixture and Eulerian multiphase models. In the Mixture model, a single momentum equation is solved for a mixture whose properties are determined from the sum of its phases. In the Eulerian model, a momentum equation is solved for each phase. FLUENT allows you to directly specify a momentum source for the mixture of phases (using `DEFINE_SOURCE`) when the mixture model is used, but not for the Eulerian model. For the latter case, you can specify momentum sources for the individual phases. Hence, the multiphase model, as well as the term being modified by the UDF, determines which domain or thread is required.

For a given `DEFINE` macro used for a multiphase UDF model, the particular thread or domain pointer that gets passed depends on which phase (or mixture) the function is hooked to the solver via the graphical user interface. UDFs that are hooked to the mixture of phases are passed superdomain (or mixture-level) structures, while functions that are hooked to a particular phase are passed subdomain (or phase-level) structures. `DEFINE_ADJUST` and `DEFINE_INIT` UDFs are hardwired to the mixture-level domain. For your convenience, Tables 1.10.1–1.10.6 summarize each multiphase model, and the phase on which UDFs are specified for the given variables. From this information, you can infer which domain structure is passed from the solver to the UDF. If your UDF is not explicitly passed a pointer to the thread or domain structure that it requires, you will need to use a special multiphase-specific utility (e.g., `THREAD_SUB_THREAD`) to retrieve it. (See Chapter 6 for details.)
### 1.10 Special Considerations for Multiphase UDFs

#### Table 1.10.1: DEFINE Macro Usage for the VOF Model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Phase Specified On</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume fraction</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>velocity at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>pressure at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>boundary temperature</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent kinetic energy</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent dissipation rate</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>mass source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>momentum source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>energy source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent kinetic energy source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent dissipation rate source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>density</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
</tbody>
</table>
### Table 1.10.2: DEFINE Macro Usage for the Mixture Model

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Phase Specified On</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume fraction</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>velocity at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>pressure at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>boundary temperature</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent kinetic energy</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent dissipation rate</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>mass source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>momentum source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>energy source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent kinetic energy source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent dissipation rate source</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>density</td>
<td>DEFINEPROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>diameter</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>slip velocity</td>
<td>DEFINEVECTOR_</td>
<td>phase interaction</td>
</tr>
<tr>
<td></td>
<td>EXCHANGE_PROPERTY</td>
<td></td>
</tr>
</tbody>
</table>
Table 1.10.3: **DEFINE** Macro Usage for the Eulerian Model (Laminar Flow)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Phase Specified On</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume fraction</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>velocity at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>pressure at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>temperature at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular temperature</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>mass source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>momentum source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>energy source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>density</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular diameter</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular bulk viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular frictional viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>drag coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
<tr>
<td>lift coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
</tbody>
</table>
Table 1.10.4: DEFINE Macro Usage for the Eulerian Model (Mixture Turbulence Model)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Phase Specified On</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume fraction</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>velocity at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>pressure at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>temperature at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular temperature</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>turbulent kinetic energy</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent dissipation rate</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>mass source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>momentum source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>energy source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>turbulent kinetic energy</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>turbulent dissipation rate</td>
<td>DEFINE_SOURCE</td>
<td>mixture</td>
</tr>
<tr>
<td>density</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular diameter</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular bulk viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular frictional viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>drag coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
<tr>
<td>lift coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
</tbody>
</table>
Table 1.10.5: **DEFINE** Macro Usage for the Eulerian Model (Dispersed Turbulence Model)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Phase Specified On</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume fraction</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>velocity at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>pressure at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>temperature at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular temperature</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>turbulent kinetic energy</td>
<td>DEFINE_PROFILE</td>
<td>primary phase</td>
</tr>
<tr>
<td>turbulent dissipation rate</td>
<td>DEFINE_PROFILE</td>
<td>primary phase</td>
</tr>
<tr>
<td>mass source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>momentum source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>energy source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>turbulent kinetic energy source</td>
<td>DEFINE_SOURCE</td>
<td>primary phase</td>
</tr>
<tr>
<td>turbulent dissip. rate source</td>
<td>DEFINE_SOURCE</td>
<td>primary phase</td>
</tr>
<tr>
<td>density</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular diameter</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular bulk viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular frictional viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>drag coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
<tr>
<td>lift coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
</tbody>
</table>
Table 1.10.6: **DEFINE** Macro Usage for the Eulerian Model (Per-Phase Turbulence Model)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Macro</th>
<th>Phase Specified On</th>
</tr>
</thead>
<tbody>
<tr>
<td>volume fraction</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>velocity at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>pressure at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>mixture</td>
</tr>
<tr>
<td>temperature at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular temperature</td>
<td>DEFINE_PROFILE</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>turbulent kinetic energy</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>turbulent dissipation rate</td>
<td>DEFINE_PROFILE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>mass source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>momentum source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>energy source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>turbulent kinetic energy source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>turbulent dissip. rate source</td>
<td>DEFINE_SOURCE</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>density</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>primary and secondary phase(s)</td>
</tr>
<tr>
<td>granular diameter</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular bulk viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>granular frictional viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>secondary phase(s)</td>
</tr>
<tr>
<td>drag coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
<tr>
<td>lift coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>phase interaction</td>
</tr>
</tbody>
</table>
Chapter 2.  C Programming Basics for UDFs

This chapter contains an overview of C programming basics for UDFs.

- Section 2.1: Introduction
- Section 2.2: Commenting Your C Code
- Section 2.3: C Data Types in FLUENT
- Section 2.4: Constants
- Section 2.5: Variables
- Section 2.6: User-Defined Data Types
- Section 2.7: Casting
- Section 2.8: Functions
- Section 2.9: Arrays
- Section 2.10: Pointers
- Section 2.11: Control Statements
- Section 2.12: Common C Operators
- Section 2.13: C Library Functions
- Section 2.14: Macro Substitution Directive Using \#define
- Section 2.14: File Inclusion Directive Using \#include
- Section 2.15: Comparison with FORTRAN

2.1 Introduction

This chapter contains some basic information about the C programming language that may be helpful when writing UDFs in FLUENT. It is not intended to be used as a primer on C and assumes that you have some programming experience. There are many topics and details that are not covered in this chapter including, for example, while and do-while control statements, unions, recursion, structures, and reading and writing files.

If you are unfamiliar with C or need information on C programming, consult any one of the available resource books on the subject (e.g., [2, 3]).
2.2 Commenting Your C Code

It is good programming practice to document your C code with comments that are useful for explaining the purpose of the function. In a single line of code, your comments must begin with the /* identifier, followed by text, and end with the */ identifier as shown by the following:

/* This is how I put a comment in my C program */

Comments that span multiple lines are bracketed by the same identifiers:

/* This is how I put a comment in my C program
   that spans more
   than one line */

2.3 C Data Types in FLUENT

The UDF interpreter in FLUENT supports the following standard C data types:

- `int` integer number
- `long` integer number of increased range
- `float` floating point (real) number
- `double` double-precision floating point (real) number
- `char` single byte of memory, enough to hold a character

Note that in FLUENT, `real` is a typedef that switches between `float` for single-precision arithmetic, and `double` for double-precision arithmetic. Since the interpreter makes this assignment automatically, it is good programming practice to use the `real` typedef when declaring all `float` and `double` data type variables in your UDF.

2.4 Constants

Constants are absolute values that are used in expressions and need to be defined in your C program using `#define`. Simple constants are decimal integers (e.g., 0, 1, 2). Constants that contain decimal points or the letter e are taken as floating point constants. As a convention, constants are typically declared using all capitals. For example, you may set the ID of a zone, or define constants `YMIN` and `YMAX` as shown below:

```c
#define WALL_ID 5
#define YMIN 0.0
#define YMAX 0.4064
```
2.5 Variables

A variable (or object) is a place in memory where you can store a value. Every variable has a type (e.g., real), a name, and a value, and may have a storage class identifier (static or extern). All variables must be declared before they can be used. By declaring a variable ahead of time, the C compiler knows what kind of storage to allocate for the value.

Global variables are variables that are defined outside of any single function and are visible to all function(s) within a UDF source file. Global variables can also be used by other functions outside of the source file unless they are declared as static (see Section 2.5.3). Global variables are typically declared at the beginning of a file, after preprocessor directives as in

```
#include "udf.h"

real volume; /* real variable named volume is declared globally */
```

Local variables are variables that are used in a single function. They are created when the function is called, and are destroyed when the function returns unless they are declared as static (see Section 2.5.3). Local variables are declared within the body of a function (inside the curly braces {}). In the example below, mu_lam and temp are local variables. The value of these variables is not preserved once the function returns.

```
DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
    real mu_lam; /* local variable */
    real temp = C_T(cell, thread); /* local variable */

    if (temp > 288.)
        mu_lam = 5.5e-3;
    else if (temp > 286.)
        mu_lam = 143.2135 - 0.49725 * temp;
    else
        mu_lam = 1.;

    return mu_lam;
}
```
2.5.1 Declaring Variables

A variable declaration begins with the data type (e.g., int), followed by the name of one or more variables of the same type that are separated by commas. A variable declaration can also contain an initial value, and always ends with a semicolon (;). Variable names must begin with a letter in C. A name can include letters, numbers, and the underscore (_) character. Note that the C preprocessor is case-sensitive (recognizes uppercase and lowercase letters as being different). Below are some examples of variable declarations.

```c
int n; /* declaring variable n as an integer */
int i1, i2; /* declaring variables i1 and i2 as integers */
float tmax = 0.; /* tmax is a floating point real number that is initialized to 0 */
real average_temp = 0.0; /* average_temp is a real number initialized to 0.0 */
```

2.5.2 External Variables

If you have a global variable that is declared in one source code file, but a function in another source file needs to use it, then it must be defined in the other source file as an external variable. To do this, simply precede the variable declaration by the word extern as in

```c
extern real volume;
```

If there are several files referring to that variable then it is convenient to include the extern definition in a header (.h) file, and include the header file in all of the .c files that want to use the external variable. Only one .c file should have the declaration of the variable without the extern keyword. Below is an example that demonstrates the use of a header file.

```c
extern can be used only in compiled UDFs.
```
Example

Suppose that there is a global variable named `volume` that is declared in a C source file named `file1.c`

```c
#include "udf.h"
real volume;    /* real variable named volume is declared globally */

DEFINE_ADJUST(compute_volume, domain)
{
   /* code that computes volume of some zone */
   volume = ....
}
```

If multiple source files want to use `volume` in their computations, then `volume` can be declared as an external variable in a header file (e.g., `extfile.h`)

```c
/* extfile.h
   Header file that contains the external variable declaration for
   volume */
extern real volume;
```

Now another file named `file2.c` can declare `volume` as an external variable by simply including `extfile.h`.

```c
/* file2.c
   #include "udf.h"
   #include "extfile.h" /* header file containing extern declaration is included */

DEFINE_SOURCE(heat_source,c,t,ds,eqn)
{
   /* code that computes the per unit volume source using the total
      volume computed in the compute_volume function from file1.c */

   real total_source = ...;
   real source;

   source = total_source/volume;
   return source;
}
```
2.5.3 Static Variables

The static operator has different effects depending on whether it is applied to local or global variables. When a local variable is declared as static the variable is prevented from being destroyed when a function returns from a call. In other words, the value of the variable is preserved. When a global variable is declared as static the variable is “file global”. It can be used by any function within the source file in which it is declared, but is prevented from being used outside the file, even if is declared as external. Functions can also be declared as static. A static function is visible only to the source file in which it is defined.

! static variables and functions can be declared only in compiled UDF source files.

Example - Static Global Variable

/* mysource.c */
#include "udf.h"

static real abs_coeff = 1.0; /* static global variable */
    /* used by both functions in this source file but is
     * not visible to the outside */

#define SOURCE(energy_source, c, t, dS, eqn)
{
    real source;   /* local variable
    int P1 = ....;  /* local variable
        value is not preserved when function returns */

    dS[eqn] = -16.* abs_coeff * SIGMA_SBC * pow(C_T(c,t),3.);
    source =-abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
    return source;
}

#define SOURCE(p1_source, c, t, dS, eqn)
{
    real source;
    int P1 = ....;
    dS[eqn] = -abs_coeff;
    source = abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
    return source;
}
2.6 User-Defined Data Types

C also allows you to create user-defined data types using structures and typedef. (For information about structures in C, see [2].) An example of a structured list definition is shown below.

! typedef can only be used for compiled UDFs.

Example

```c
typedef struct list{int a;
    real b;
    int c;} mylist; /* mylist is type structure list */
mylist x,y,z;       /* x,y,z are type structure list */
```

2.7 Casting

You can convert from one data type to another by casting. A cast is denoted by type, where the type is int, float, etc., as shown in the following example:

```c
int x = 1;
real y = 3.14159;
int z = x+((int) y); /* z = 4 */
```

2.8 Functions

Functions perform tasks. Tasks may be useful to other functions defined within the same source code file, or they may be used by a function external to the source file. A function has a name (that you supply) and a list of zero or more arguments that are passed to it. A function has a body enclosed within curly braces that contains instructions for carrying out the task. A function may return a value of a particular type. C functions pass data by value.

Functions either return a value of a particular data type (e.g., real), or do not return any value if they are of type void. To determine the return data type for the DEFINE macro you will use to define your UDF, look at the macro’s corresponding #define statement in the udf.h file or see Appendix A for a listing.

! C functions cannot alter their arguments. They can, however, alter the variables that their arguments point to.
2.9 Arrays

Arrays of variables can be defined using the notation \texttt{name[size]}, where \texttt{name} is the variable name and \texttt{size} is an integer that defines the number of elements in the array. The index of a C array always begins at 0.

Arrays of variables can be of different data types as shown below.

Examples

\begin{verbatim}
int a[10], b[10][10];
real radii[5];

a[0] = 1; /* a 1-Dimensional array of variable a */
radii[4] = 3.14159265; /* a 1-Dimensional array of variable radii */
b[10][10] = 4; /* a 2-Dimensional array of variable b */
\end{verbatim}

2.10 Pointers

A pointer is a variable that contains an address in memory where the value referenced by the pointer is stored. In other words, a pointer is a variable that points to another variable by referring to the other variable's address. Pointers contain memory addresses, not values. Pointer variables must be declared in C using the \texttt{*} notation. Pointers are widely used to reference data stored in structures and to pass data among functions (by passing the addresses of the data).

For example,

\begin{verbatim}
int *ip;
\end{verbatim}

declares a pointer named \texttt{ip} that points to an integer variable. Now suppose you want to assign an address to pointer \texttt{ip}. To do this, you can use the \& notation. For example,

\begin{verbatim}
ip = &a;
\end{verbatim}

assigns the address of variable \texttt{a} to pointer \texttt{ip}.

You can retrieve the value of variable \texttt{a} that pointer \texttt{ip} is pointing to by

\begin{verbatim}
*ip
\end{verbatim}
Alternatively, you can set the value of the variable that pointer `ip` points. For example,

```c
*ip = 4;
```

assigns a value of 4 to the variable that pointer `ip` is pointing. The use of pointers is demonstrated by the following:

```c
int a = 1;
int *ip;
ip = &a; /* &a returns the address of variable a */
printf("content of address pointed to by ip = %d\n", *ip);
*ip = 4; /* a = 4 */
printf("now a = %d\n", a);
```

Here, an integer variable `a` is initialized to 1. Next, `ip` is declared as a pointer to an integer variable. The address of variable `a` is then assigned to pointer `ip`. Next, the integer value of the address pointed to by `ip` is printed using `*ip`. (This value is 1.) The value of variable `a` is then indirectly set to 4 using `*ip`. The new value of `a` is then printed. Pointers can also point to the beginning of an array, and are strongly connected to arrays in C.

**Pointers as Function Arguments**

C functions can access and modify their arguments through pointers. In FLUENT, thread and domain pointers are common arguments to UDFs. When you specify these arguments in your UDF, the FLUENT solver automatically passes data that the pointers are referencing to your UDF so that your function can access solver data. (You do not have to declare pointers that are passed as arguments to your UDF from the solver.) For example, one of the arguments passed to a UDF that specifies a custom profile (defined by the `DEFINE_PROFILE` macro) is the pointer to the thread applied to by the boundary condition. The `DEFINE_PROFILE` function accesses the data pointed to by the thread pointer.
2.11 Control Statements

You can control the order in which statements are executed in your C program using control statements like if, if-else, and for loops. Control statements make decisions about what is to be executed next in the program sequence.

2.11.1 if Statement

An if statement is a type of conditional control statement. The format of an if statement is:

```c
if (logical-expression)
    {statements}
```

where `logical-expression` is the condition to be tested, and `statements` are the lines of code that are to be executed if the condition is met.

Example

```c
if (q != 1)
    {a = 0; b = 1;}
```

2.11.2 if-else Statement

if-else statements are another type of conditional control statement. The format of an if-else statement is:

```c
if (logical-expression)
    {statements}
else
    {statements}
```

where `logical-expression` is the condition to be tested, and the first set of `statements` are the lines of code that are to be executed if the condition is met. If the condition is not met, then the statements following `else` are executed.
Example

```c
if (x < 0.)
  y = x/50.;
else
  {
    x = -x;
    y = x/25.;
  }
```

The equivalent FORTRAN code is shown below for comparison.

```fortran
IF (X.LT.0.) THEN
  Y = X/50.
ELSE
  X = -X
  Y = X/25.
ENDIF
```

### 2.11.3 for Loops

for loops are control statements that are a basic looping construct in C. They are analogous to do loops in FORTRAN. The format of a for loop is

```c
for (begin ; end ; increment)
  {statements}
```

where `begin` is the expression that is executed at the beginning of the loop; `end` is the logical expression that tests for loop termination; and `increment` is the expression that is executed at the end of the loop iteration (usually incrementing a counter).

Example

```c
/* Print integers 1-10 and their squares */

int i, j, n = 10;

for (i = 1 ; i <= n ; i++)
  {
    j = i*i;
    printf("%d %d\n",i,j);
  }
```
The equivalent FORTRAN code is shown below for comparison.

    INTEGER I,J
    N = 10
    DO I = 1,10
      J = I*I
      WRITE (*,*) I,J
    ENDDO

2.12 Common C Operators

Operators are internal C functions that, when they are applied to values, produce a result. Common types of C operators are arithmetic and logical.

2.12.1 Arithmetic Operators

Some common arithmetic operators are listed below.

    =    assignment
    +    addition
    -    subtraction
    *    multiplication
    /    division
    %    modulo reduction
    ++   increment
    --   decrement

Note that multiplication, division, and modulo reduction (%) operations will be performed before addition and subtraction in any expression. When division is performed on two integers, the result is an integer with the remainder discarded. Modulo reduction is the remainder from integer division. The ++ operator is a shorthand notation for the increment operation.

2.12.2 Logical Operators

Some common logical operators are listed below.

    <    less than
    <=   less than or equal to
    >    greater than
    >=   greater than or equal to
    ==   equal to
    !=   not equal to
2.13 C Library Functions

C compilers include a library of standard mathematical and I/O functions that you can use when you write your UDF code. Lists of standard C library functions are presented in the following sections. Definitions for standard C library functions can be found in various header files (e.g., global.h). These header files are all included in the udf.h file.

2.13.1 Trigonometric Functions

The trigonometric functions shown below are computed (with one exception) for the variable \( x \). Both the function and the argument are double-precision real variables. The function \( \text{acos}(x) \) is the arccosine of the argument \( x \), \( \cos^{-1}(x) \). The function \( \text{atan2}(x, y) \) is the arctangent of \( x/y \), \( \tan^{-1}(x/y) \). The function \( \text{cosh}(x) \) is the hyperbolic cosine function, etc.

\[
\begin{align*}
\text{double acos (double x); } & \text{ returns the arccosine of } x \\
\text{double asin (double x); } & \text{ returns the arcsine of } x \\
\text{double atan (double x); } & \text{ returns the arctangent of } x \\
\text{double atan2 (double x, double y); } & \text{ returns the arctangent of } x/y \\
\text{double cos (double x); } & \text{ returns the cosine of } x \\
\text{double sin (double x); } & \text{ returns the sine of } x \\
\text{double tan (double x); } & \text{ returns the tangent of } x \\
\text{double cosh (double x); } & \text{ returns the hyperbolic cosine of } x \\
\text{double sinh (double x); } & \text{ returns the hyperbolic sine of } x \\
\text{double tanh (double x); } & \text{ returns the hyperbolic tangent of } x
\end{align*}
\]

2.13.2 Miscellaneous Mathematical Functions

The C functions shown on the left below correspond to the mathematical functions shown on the right.

\[
\begin{align*}
\text{double sqrt (double x); } & \sqrt{x} \\
\text{double pow(double x, double y); } & x^y \\
\text{double exp (double x); } & e^x \\
\text{double log (double x); } & \ln(x) \\
\text{double log10 (double x); } & \log_{10}(x) \\
\text{double fabs (double x); } & |x| \\
\text{double ceil (double x); } & \text{smallest integer not less than } x \\
\text{double floor (double x); } & \text{largest integer not greater than } x
\end{align*}
\]
2.13.3 Standard I/O Functions

A number of standard input and output (I/O) functions are available in C and in FLUENT. They are listed below. All of the functions work on a specified file except for *printf*, which displays information that is specified in the argument of the function. The format string argument is the same for *printf*, *fprintf*, and *fscanf*. Note that all of these standard C I/O functions are supported by the interpreter, so you can use them in either interpreted or compiled UDFs. For more information about standard I/O functions in C, you should consult a reference guide (e.g., [2]).

**Common C I/O Functions**

- `fopen("filename", "mode");` opens a file
- `fclose(fp);` closes a file
- `printf("format", ...);` formatted print to the console
- `fprintf(fp, "format", ...);` formatted print to a file
- `fscanf(fp, "format", ...);` formatted read from a file

It is not possible to use the *scanf* C function in FLUENT.

**fopen**

```c
FILE *fopen(char *filename, char *mode);
```

The function *fopen* opens a file in the mode that you specify. It takes two arguments: `filename` and `mode`. `filename` is a pointer to the file you want to open. `mode` is the mode in which you want the file opened. The options for `mode` are read "r", write "w", and append "a". Both arguments must be enclosed in quotes. The function returns a pointer to the file that is to be opened.

Before using *fopen*, you will first need to define a local pointer of type `FILE` that is defined in `stdio.h` (e.g., `fp`). Then, you can open the file using *fopen*, and assign it to the local pointer as shown below. Recall that `stdio.c` is included in the `udf.h` file, so you don't have to include it in your function.

```c
FILE *fp; /* define a local pointer fp of type FILE */
fp = fopen("data.txt","r"); /* open a file named data.txt in read-only mode and assign it to fp */
```
2.13 C Library Functions

fclose

int fclose(FILE *fp);

The function fclose closes a file that is pointed to by the local pointer passed as an argument (e.g., fp).

fclose(fp); /* close the file pointed to by fp */

printf

int printf(char *format, ...);

The function printf is a general-purpose printing function that prints to the console in a format that you specify. The first argument is the format string. It specifies how the remaining arguments are to be displayed in the console window. The format string is defined within quotes. The value of the replacement variables that follow the format string will be substituted in the display for all instances of %type. The % character is used to designate the character type. Some common format characters are: %d for integers, %f for floating point numbers, and %e for floating point numbers in exponential format (with e before the exponent). The format string for printf is the same as for fprintf and fscanf.

In the example below, the text Content of variable a is: will be displayed in the console window, and the value of the replacement variable, a, will be substituted in the message for all instances of %d.

Example:

int a = 5;
printf("Content of variable a is: %d\n", a); /* \n denotes a new line */

! (UNIX only) It is recommended that you use the Fluent Inc. Message utility instead of printf for compiled UDFs. See Section 6.9 for details on the Message macro.

fprintf

int fprintf(FILE *fp, char *format, ...);

The function fprintf writes to a file that is pointed to by fp, in a format that you specify. The first argument is the format string. It specifies how the remaining arguments are to be written to the file. The format string for fprintf is the same as for printf and fscanf.
Example:

FILE *fp;
fprintf(fp,"%12.4e %12.4e %5d\n",x_array[j][0], x_array[j][1], noface);

int data1 = 64.25;
int data2 = 97.33;
fprintf(fp, "%4.2d %4.2d\n", data1, data2);

fscanf

int fscanf(FILE *fp, char *format, ...);

The function fscanf reads from a file that is pointed to by fp, in a format that you specify. The first argument is the format string. It specifies how the data that is to be read is to be interpreted. The replacement variables that follow the format string are used to store values that are read. The replacement variables are preceded by the & character. Note that the format string for fscanf is the same as for fprintf and printf.

In the example below, two floating point numbers are read from the file pointed to by fp, and are stored in the variables f1 and f2.

Example:

FILE *fp;
fscanf(fp, "%f %f", &f1, &f2);

! You cannot use the scanf I/O function in FLUENT. You must use fscanf instead.

2.14 Preprocessor Directives

The UDF interpreter supports C preprocessor directives including #define and #include.

Macro Substitution Directive Using #define

When you use the #define macro substitution directive, the C preprocessor (e.g., cpp) performs a simple substitution and expands the occurrence of each argument in macro using the replacement-text.

#define  macro  replacement-text

For example, the macro substitution directive given by

#define RAD 1.2345
will cause the preprocessor to replace all instances of the variable \texttt{RAD} in your UDF with the number \texttt{1.2345}. There may be many references to the variable \texttt{RAD} in your function, but you only have to define it once in the macro directive; the preprocessor does the work of performing the substitution throughout your code.

In another example

\begin{verbatim}
#define AREA_RECTANGLE(X,Y) ((X)*(Y))
\end{verbatim}

all of the references to \texttt{AREA_RECTANGLE(X,Y)} in you UDF are replaced by the product of \texttt{(X)} and \texttt{(Y)}.

\section*{File Inclusion Directive Using \texttt{#include}}

When you use the \texttt{#include} file inclusion directive, the C preprocessor replaces the line \texttt{#include filename} with the contents of the named file.

\begin{verbatim}
#include "filename"
\end{verbatim}

The file you name must reside in your current directory. The only exception to this rule is the \texttt{udf.h} file, which is read automatically by the FLUENT solver.

For example, the file inclusion directive given by

\begin{verbatim}
#include "udf.h"
\end{verbatim}

will cause the \texttt{udf.h} file to be included with your source code. The FLUENT solver automatically reads the \texttt{udf.h} file from the Fluent.Inc/fluents6.x/src/ directory.

\section*{2.15 Comparison with FORTRAN}

Many simple C functions are similar to FORTRAN function subroutines as shown in the example below:

\begin{verbatim}
A simple C function

int myfunction(int x)
{
    int x,y,z;
    y = 11;
    z = x+y;
    printf("z = %d",z);
    return z;
}

An equivalent FORTRAN function

INTEGER FUNCTION MYFUNCTION(X)

    INTEGER X,Y,Z
    Y = 11
    Z = X+Y
    WRITE (*,100) Z
    MYFUNCTION = Z
END
\end{verbatim}
Chapter 3. A Step-by-Step UDF Example

This chapter contains an example that utilizes a 7-step process to code a UDF and use it effectively in your FLUENT model. It begins with an overview of the process.

1. Define your problem. (Section 3.1.1)

2. Create a C source code file. (Section 3.1.2)

3. Start FLUENT and read in (or set up) the case file. (Section 3.1.3)

4. Interpret the source file. (Section 3.1.4)

5. Hook the UDF to FLUENT. (Section 3.1.5)

6. Run the calculation. (Section 3.1.6)

7. Analyze the numerical solution and compare it to expected results. (Section 3.1.7)

3.1 Process Overview

To begin the process, you'll need to define the problem you wish to solve using a UDF (Step 1). For example, suppose you want to use a UDF to define a customized boundary profile for your problem. You will first need to define the set of mathematical equation(s) that describe the profile.

Next you will need to translate the mathematical equations (conceptual design) into a function written in C (Step 2). You can do this using any text editor. Save the file with a .c suffix (e.g., velocity_profile.c) in your working directory.

Once you have written your C function, you are ready to start FLUENT and read in (or set up) your case file (Step 3). You will then need to interpret, or compile and load your source file, debug the source file (Step 4), and then hook the UDF to FLUENT (Step 5). Finally you’ll run the calculation (Step 6), analyze the results from your simulation, and compare them to expected results (Step 7). You may loop through this entire process more than once, depending on the results of your analysis. Follow the step-by-step process in the sections below to see how this is done.
3.1.1 Step 1: Define Your Problem

The first step in creating a UDF and using it in your FLUENT model involves defining your model equations.

Consider the turbine vane illustrated in Figure 3.1.1. An unstructured grid is used to model the flow field surrounding the vane. The domain extends from a periodic boundary on the bottom to an identical one on the top, a velocity inlet on the left, and a pressure outlet on the right.

![Turbine Vane Grid](image)

Figure 3.1.1: The Grid for the Turbine Vane Example

A flow field in which a constant $x$ velocity is applied at the inlet will be compared with one where a parabolic $x$ velocity profile is applied. The results of a constant-velocity applied field (of 20 m/s) at the inlet are shown in Figures 3.1.2 and 3.1.3. The initial constant-velocity field is distorted as the flow moves around the turbine vane.

Now suppose that you want to impose a non-uniform $x$ velocity to the turbine vane inlet, which is described by the profile

\[ v_x = 20 - 20 \left( \frac{y}{0.0745} \right)^2 \]

where the variable $y$ is 0.0 at the center of the inlet, and extends to values of $\pm 0.0745$ m at the top and bottom. Thus the $x$ velocity will be 20 m/s at the center of the inlet, and 0 at the edges.

To solve this type of problem, you can write a UDF to describe the profile function, and then apply it to your FLUENT model.
3.1 Process Overview

Figure 3.1.2: Velocity Magnitude Contours for a Constant Inlet \( x \) Velocity

Figure 3.1.3: Velocity Vectors for a Constant Inlet \( x \) Velocity
3.1.2 Step 2: Create a C Source File

Now that you have determined the equation that defines your UDF, you can use any text editor to create a text file containing C code. Save the source code file with a .c extension (e.g., udfexample.c) in your working directory. The following source code listing contains only a single UDF. Your source file can contain multiple concatenated functions.

Below is an example of how the equation can be implemented in a UDF. The functionality of the UDF is designated by the leading DEFINE macro. Here, the DEFINE_PROFILE macro is used to indicate to the solver that the following code will provide profile information at boundaries. Other DEFINE macros will be discussed later in this manual.

```c
#include "udf.h" /* must be at the beginning of every UDF you write */

DEFINE_PROFILE(inlet_x_velocity, thread, index)
{
    real x[ND_ND]; /* this will hold the position vector */
    real y;
    face_t f;

    begin_f_loop(f, thread) /* loops over all faces in the thread passed in the DEFINE macro argument */
    {
        F_CENTROID(x,f,thread);
        y = x[1];
        F_PROFILE(f, thread, index) = 20. - y*y/(.0745*.0745)*20.;
    }
    end_f_loop(f, thread)
}
```

The first argument of the DEFINE_PROFILE macro, inlet_x_velocity, is used to identify the function in the Velocity Inlet panel. The name is arbitrary and is specified by you. The equation in the function will be applied to all cell faces (identified by f in the face loop) on a given boundary zone (identified by thread). The thread is defined automatically when you select the UDF for a particular boundary in the FLUENT graphical user-interface. The index is defined automatically through the begin_f_loop utility. In this UDF, the begin_f_loop macro is used to loop through all cell faces in the boundary zone. For each face, the coordinates of the face centroid are accessed by the F_CENTROID macro.
The $y$ coordinate $y$ is used in the parabolic profile equation and the returned velocity is assigned to the face through the $F_{\text{PROFILE}}$ macro. $\text{begin}_f\_\text{loop}$ and $F_{\text{PROFILE}}$ are Fluent-supplied macros. See Chapter 5 for details.

### 3.1.3 Step 3: Start FLUENT and Read (or Set Up) the Case File

Once you have created your UDF, you are ready to begin the problem setup in FLUENT.

1. Start FLUENT from your working directory.
2. Read (or set up) your case file.

### 3.1.4 Step 4: Interpret the Source File

You are now ready to interpret the UDF that was created in Step 2, and is contained within the source file named $\text{udfexample.c}$. (Note that this step does not apply to Windows parallel networks. See Section 7.2 for details.)

1. Open the Interpreted UDFs panel.
   
   ![Interpreted UDFs Panel]

   Figure 3.1.4: The Interpreted UDFs Panel

2. Specify the source file by either typing the Source File Name or clicking Browse... to select the UDF source file name from the appropriate directory in the Select File panel. Highlight the directory path under Directories, the desired file (e.g., $\text{udfexample.c}$) under Files, and click OK.
3. Specify the C preprocessor to be used in the CPP Command Name field. You can keep the default `cpp` or you can select Use Contributed CPP to use the preprocessor supplied by Fluent Inc.

4. Keep the default Stack Size setting of 10000, unless the number of local variables in your function will cause the stack to overflow. In this case, set the Stack Size to a number that is greater than the number of local variables used.

5. Click Display Assembly Listing if you would like a listing of assembly language code to appear in your console window as your function is interpreted.

6. Click Interpret to interpret your UDF.

```assembly
inlet_x_velocity:
    .local.pointer thread (r0)
    .local.int nv (r1)
0    .local.end
0    save
    .local.int f (r3)
1    push.int 0
    .local.pointer x (r4)
3    begin.data 8 bytes, 0 bytes initialized:
7    save
.
.
.
156    pre.inc.int f (r3)
158    pop.int
159    b .L3 (22)
.L2:
161    restore
162    restore
163    ret.v
```

Note that if your compilation is unsuccessful, then FLUENT will report an error and you will need to debug your program. See Section 7.2.1 for details.

7. Click Close when the interpreter has finished.

8. Write the case file. The interpreted UDF, named `inlet_x_velocity`, will be saved with the case file so that the function will be automatically interpreted whenever the case is subsequently read. If the Display Assembly Listing option was chosen, then the assembly code will appear in the console window upon reading.
3.1 Process Overview

In addition to interpreting this particular example profile UDF, you can also compile it using the Compiled UDFs panel in FLUENT. See Section 7.3 for details on compiling UDFs using the graphical user interface.

3.1.5 Step 5: Hook Your UDF to FLUENT

Now that you have interpreted your UDF following the method outlined in Step 4, you are ready to hook the function to FLUENT using a graphical user interface panel. For this particular profile UDF example, the name of the function will appear in a drop-down list in the Velocity Inlet boundary condition panel.

Define → Boundary Conditions...

In the X-Velocity drop-down list, select udf inlet x_velocity, the name that was given to the function in our sample problem. Once selected, the UDF will be used in the calculation, rather than the value of (in this example) 0 that appears in the X-Velocity field. Click OK to accept the new boundary condition and close the panel.

3.1.6 Step 6: Run the Calculation

Run the calculation as usual.

Solve → Iterate...
3.1.7 Step 7: Analyze the Numerical Solution and Compare to Expected Results

Once the solution is run to convergence, obtain a revised velocity field. The velocity magnitude contours for the parabolic inlet x velocity are shown in Figure 3.1.5, and can be compared to the results of a constant-velocity field of 20 m/sec (Figure 3.1.2). For the constant velocity condition, the flow field is distorted as the flow moves around the turbine vane. The velocity field for the imposed parabolic profile, however, shows a maximum at the center of the inlet, which drops to zero at the edges.

Figure 3.1.5: Velocity Magnitude Contours for a Parabolic Inlet x Velocity
Chapter 4. **DEFINE Macros**

This chapter contains descriptions of predefined macros provided by Fluent Inc. that you will need to use to define your UDF. These macros are referred to as **DEFINE** macros.

The chapter is organized in the following sections:

- Section 4.1: Introduction
- Section 4.2: General **DEFINE** Macros
- Section 4.3: Model-Specific **DEFINE** Macros
- Section 4.4: Multiphase **DEFINE** Macros
- Section 4.5: Dynamic Mesh **DEFINE** Macros
- Section 4.6: DPM **DEFINE** Macros

### 4.1 Introduction

**DEFINE** macros have been grouped into the following five categories:

- general solver
- model-specific
- multiphase
- dynamic mesh model
- discrete phase model (DPM)

For each of the **DEFINE** macros listed in this chapter, a source code example of a UDF that uses it is provided, where available. Many of the examples make extensive use of other macros discussed in later chapters on solver access (Chapter 5) and utilities (Chapter 6). Note that not all of the examples in the chapter are complete functions that can be executed as stand-alone functions in FLUENT. Examples are intended to demonstrate **DEFINE** macro usage only.

Definitions for all **DEFINE** macros are contained in the `udf.h` header file. For your convenience, they are listed in Appendix A.

Note that all of the arguments to a **DEFINE** macro need to be placed on the same line in your source code. Splitting the **DEFINE** statement onto several lines will result in a compilation error.
4.2 General DEFINE Macros

The DEFINE macros presented in this section implement general solver functions that are model-independent in FLUENT. Table 4.2.1 provides a quick reference guide to the DEFINE macros, the functions they are used to define, and the panels where they are activated in FLUENT. Definitions of each DEFINE macro are contained in the udf.h header file and are provided in Appendix A.

- DEFINE_ADJUST (Section 4.2.1)
- DEFINE_EXECUTE_AT_END (Section 4.2.2)
- DEFINE_INIT (Section 4.2.3)
- DEFINE_ON_DEMAND (Section 4.2.4)
- DEFINE_RW_FILE (Section 4.2.5)

Table 4.2.1: Quick Reference Guide for General Solver DEFINE Macros

<table>
<thead>
<tr>
<th>Function</th>
<th>DEFINE Macro</th>
<th>Panel Activated In</th>
</tr>
</thead>
<tbody>
<tr>
<td>manipulates variables</td>
<td>DEFINE_ADJUST</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>executes at end of iteration or time step</td>
<td>DEFINE_EXECUTE_AT_END</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>initializes variables</td>
<td>DEFINE_INIT</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>executes asynchronously</td>
<td>DEFINE_ON_DEMAND</td>
<td>Execute On Demand</td>
</tr>
<tr>
<td>reads/writes variables to case and data files</td>
<td>DEFINE_RW_FILE</td>
<td>User-Defined Function Hooks</td>
</tr>
</tbody>
</table>
4.2.1 DEFINE_ADJUST

Description

DEFINE_ADJUST is a general-purpose macro that can be used to adjust or modify FLUENT variables that are not passed as arguments. For example, you can use DEFINE_ADJUST to modify flow variables (e.g., velocities, pressure) and compute integrals. You can also use it to integrate a scalar quantity over a domain and adjust a boundary condition based on the result. A function that is defined using DEFINE_ADJUST executes at every iteration and is called at the beginning of every iteration before transport equations are solved. Refer to Figures 1.9.1 and 1.9.2 for an overview of the FLUENT solution process which shows when a DEFINE_ADJUST is called.

Usage

Macro: 

```
DEFINE_ADJUST (name, d)
```

Argument types: 

```
Domain *d
```

Function returns: 

```
void
```

There are two arguments to DEFINE_ADJUST: name and d. You will supply name, the name of the UDF. The passed argument, d, is a pointer to the domain over which the adjust function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the domain pointer that is passed to the function by the solver is the mixture-level domain pointer.
Example 1

The following UDF, named `my_adjust`, integrates the turbulent dissipation over the entire domain using `DEFINE_ADJUST`. This value is then printed to the console window. The UDF is called once every iteration. It can be executed as an interpreted or compiled UDF in FLUENT.

```c
#include "udf.h"

DEFINE_ADJUST(my_adjust, d)
{
    Thread *t;
    /* Integrate dissipation. */
    real sum_diss=0.;
    cell_t c;

    thread_loop_c (t,d)
    {
        begin_c_loop (c,t)
            sum_diss += C_D(c,t)*
                C_VOLUME(c,t);
        end_c_loop (c,t)
    }

    printf("Volume integral of turbulent dissipation: %g\n", sum_diss);
}
```
Example 2

The following UDF, named adjust_fcn, specifies a user-defined scalar as a function of the gradient of another user-defined scalar, using DEFINE_ADJUST. The function is called once every iteration. It is executed as a compiled UDF in FLUENT.

```c
#include "udf.h"

DEFINE_ADJUST(adjust_fcn, d)
{
    Thread *t;
    cell_t c;
    real K_EL = 1.0;

    /* Do nothing if gradient isn’t allocated yet. */
    if (! Data_Valid_P())
        return;

    thread_loop_c (t,d)
    {
        if (FLUID_THREAD_P(t))
        {
            begin_c_loop_all (c,t)
            {
                C_UDSI(c,t,1) +=
                K_EL*NV_MAG2(C_UDSI_G(c,t,0))*C_VOLUME(c,t);
            }
            end_c_loop_all (c,t)
        }
    }
}
```

Hooking an Adjust UDF to FLUENT

After the UDF that you have defined using DEFINE_ADJUST is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.1.1 for details on how to hook your DEFINE_ADJUST UDF to FLUENT.
4.2.2 DEFINE_EXECUTE_AT_END

Description

`DEFINE_EXECUTE_AT_END` is a general purpose macro that is executed at the end of an iteration in a steady state run, or at the end of a time step in a transient run. You can use `DEFINE_EXECUTE_AT_END` when you want to calculate flow quantities at these particular times. Note that you do not have to specify whether your execute-at-end UDF gets executed at the end of a time step or the end of an iteration. This is done automatically when you select the steady or unsteady time method in your FLUENT model.

Usage

Macro: `DEFINE_EXECUTE_AT_END (name)`

Argument types: none

Function returns: void

There is only one argument to `DEFINE_EXECUTE_AT_END`: `name`. You will supply `name`, the name of the UDF.

Unlike `DEFINE_ADJUST`, `DEFINE_EXECUTE_AT_END` is not passed a domain pointer. Therefore, if your function requires access to a domain pointer, then you will need to use the utility `Get_Domain(ID)` to explicitly obtain it (see Section 6.5 and the example below).

If your UDF requires access to a phase domain pointer in a multiphase solution, then it will need to pass the appropriate phase ID to `Get_Domain` in order to obtain it.
Example

The following UDF, named execute_at_end, integrates the turbulent dissipation over the entire domain using DEFINE_EXECUTE_AT_END and prints it to the console window at the end of the current iteration or time step. It can be executed as an interpreted or compiled UDF in FLUENT.

/***************************************************************************/
UDF for integrating turbulent dissipation and printing it to
console window at the end of the current iteration or time step
***************************************************************************/

#include "udf.h"

DEFINE_EXECUTE_AT_END(execute_at_end)
{
    Domain *d;
    Thread *t;
    /* Integrate dissipation. */
    real sum_diss=0.;
    cell_t c;
    d = Get_Domain(1); /* mixture domain if multiphase */

    thread_loop_c (t,d)
    {
        if (FLUID_THREAD_P(t))
        {
            begin_c_loop (c,t)
                sum_diss += C_D(c,t) * C_VOLUME(c,t);
            end_c_loop (c,t)
        }
    }

    printf("Volume integral of turbulent dissipation: \%g\n", sum_diss);
    fflush(stdout);
}

Hooking an Execute-at-End UDF to FLUENT

After the UDF that you have defined using DEFINE_EXECUTE_AT_END is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.1.2 for details on how to hook your DEFINE_EXECUTE_AT_END UDF to FLUENT.
4.2.3 DEFINE_INIT

Description

You can use the DEFINE_INIT macro to define a set of initial values for your solution. DEFINE_INIT accomplishes the same result as patching, but does it in a different way, by means of a UDF. A DEFINE_INIT function is executed once per initialization and is called immediately after the default initialization is performed by the solver. Since it is called after the flow field is initialized, it is typically used to set initial values of flow quantities. Refer to Figures 1.9.1 and 1.9.2 for an overview of the FLUENT solution process which depicts when a DEFINE_INIT function is called.

Usage

Macro: DEFINE_INIT (name, d)

Argument types: Domain *d

Function returns: void

There are two arguments to DEFINE_INIT: name and d. You will supply name, the name of the UDF. d is a pointer to the domain over which the initialization function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the domain pointer that is passed to the function by the solver is the mixture-level domain pointer.

Example

The following UDF, named my_init_func, initializes flow field variables in a solution. It is executed once, at the beginning of the solution process. The function can be executed as an interpreted or compiled UDF in FLUENT.
UDF for initializing flow field variables

#include "udf.h"

DEFINE_INIT(my_init_func, domain)
{
    cell_t c;
    Thread *t;
    real xc[ND_ND];

    /* loop over all cell threads in the domain */
    thread_loop_c (t,domain)
    {
        /* loop over all cells */
        begin_c_loop_all (c,t)
        {
            C_CENTROID(xc,c,t);
            if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
                              pow(xc[1] - 0.5,2.),
                              pow(xc[2] - 0.5,2.))) < 0.25)
                C_T(c,t) = 400.;
            else
                C_T(c,t) = 300.;
        }
        end_c_loop_all (c,t)
    }
}

The macro ND_SUM(a,b,c) computes the sum of the first two arguments (2D) or all three arguments (3D). It is useful for writing functions involving vector operations so that the same function can be used for 2D and 3D. For a 2D case, the third argument is ignored. See Chapter 5 for a description of predefined solver access macros (e.g., C_CENTROID) and Chapter 6 for utility macros (e.g., ND_SUM).

Hooking an Initialization UDF to FLUENT

After the UDF that you have defined using DEFINE_INIT is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.1.3 for details on how to hook your DEFINE_INIT UDF to FLUENT.
4.2.4 DEFINE_ON_DEMAND

Description

You can use the DEFINE_ON_DEMAND macro to define a UDF to execute on demand in FLUENT, rather than having FLUENT call it automatically during the calculation. Your UDF will be executed immediately, once it is activated, but it is not accessible while the solver is iterating. Note that the domain pointer d is not explicitly passed as an argument to DEFINE_ON_DEMAND. Therefore, if you want to use the domain variable in your on-demand function, you will need to first retrieve it using the Get_Domain utility provided by Fluent (shown in example below). See Section 6.5 for details on Get_Domain.

Usage

Macro: DEFINE_ON_DEMAND (name)

Argument types: none

Function returns: void

There is only one argument to DEFINE_ON_DEMAND: name. You will supply name, the name of the UDF.

Example

The following UDF, named on_demand_calc, computes and prints the minimum, maximum, and average temperatures for the current data field. It then computes a temperature function

\[ f(T) = \frac{T - T_{\text{min}}}{T_{\text{max}} - T_{\text{min}}} \]

and stores it in user-defined memory location 0 (which is allocated as described in Section 5.3). Once you hook the on-demand UDF (as described in Section 8.1.4), the field values for \( f(T) \) will be available in drop-down lists in post-processing panels in FLUENT. You can select this field by choosing udm-0 in the User Defined Memory... category. If you write a data file after executing the UDF, the user-defined memory field will be saved to the data file. This source code can be interpreted or compiled in FLUENT.
UDF to calculate temperature field function and store in user-defined memory. Also print min, max, avg temperatures.

```
#include "udf.h"

DEFINE_ON_DEMAND(on_demand_calc)
{
  Domain *d; /* declare domain pointer since it is not passed as an argument to the DEFINE macro */
  real tavg = 0.;
  real tmax = 0.;
  real tmin = 0.;
  real temp, volume, vol_tot;
  Thread *t;
  cell_t c;
  d = Get_Domain(1); /* Get the domain using Fluent utility */

  /* Loop over all cell threads in the domain */
  thread_loop_c(t,d)
  {
    /* Compute max, min, volume-averaged temperature */
    /* Loop over all cells */
    begin_c_loop(c,t)
    {
      volume = C_VOLUME(c,t); /* get cell volume */
      temp = C_T(c,t); /* get cell temperature */

      if (temp < tmin || tmin == 0.) tmin = temp;
      if (temp > tmax || tmax == 0.) tmax = temp;

      vol_tot += volume;
      tavg += temp*volume;
    }
    end_c_loop(c,t)

    tavg /= vol_tot;

    printf("\n Tmin = %g  Tmax = %g  Tavg = %g\n",tmin,tmax,tavg);
  }
```

/* Compute temperature function and store in user-defined memory*/
/*(location index 0) */

define_on_demand
begin_c_loop(c,t)
{
    temp = C_T(c,t);
    C_UDMI(c,t,0) = (temp-tmin)/(tmax-tmin);
}
end_c_loop(c,t)
}

GetDomain is a macro that retrieves the pointer to a domain. It is necessary to get
the domain pointer using this macro since it is not explicitly passed as an argument
to DEFINE_ON_DEMAND. The function, named on_demand_calc, does not take any explicit
arguments. Within the function body, the variables that are to be used by the function
are defined and initialized first. Following the variable declarations, a looping macro is
used to loop over each cell thread in the domain. Within that loop another loop is used
to loop over all the cells. Within the inner loop, the total volume and the minimum,
maximum, and volume-averaged temperature are computed. These computed values are
printed to the FLUENT console. Then a second loop over each cell is used to compute
the function $f(T)$ and store it in user-defined memory location 0. Refer to Chapter 5
for a description of predefined solver access macros (e.g., C_T) and Chapter 6 for utility
macros (e.g., begin_c_loop).

Hooking an On-Demand UDF to FLUENT

After the UDF that you have defined using DEFINE_ON_DEMAND is interpreted or compiled
(see Chapter 7 for details), the name that you specified in the DEFINE macro argument
will become visible in the Execute On Demand panel in FLUENT. See Section 8.1.4 for
details on how to hook your DEFINE_ON_DEMAND UDF to FLUENT.
4.2 General DEFINE Macros

4.2.5 DEFINE_RW_FILE

Description

You can use the DEFINE_RW_FILE macro to define customized information that you want to be written to a case or data file, or read from a case or data file. You can save and restore custom variables of any data type (e.g., integer, real, Boolean, structure) using DEFINE_RW_FILE. It is often useful to save dynamic information (e.g., number of occurrences in conditional sampling) while your solution is being calculated, which is another use of this function. Note that the read order and the write order must be the same when you use this function.

Usage

Macro: 

DEFINE_RW_FILE (name, fp)

Argument types:  FILE *fp

Function returns:  void

There are two arguments to DEFINE_RW_FILE: name and fp. You will supply name, the name of the UDF. The passed argument, fp, is a pointer to the file to or from which you are writing or reading.

! Do not use the fwrite macro in DEFINE_RW_FILE functions that are running on Windows platforms. Use fprintf instead.

Example

The following C source code listing contains examples of functions that write information to a data file and read it back. These functions are concatenated into a single source file that can be interpreted or compiled in FLUENT.

/*****************************************
UDFs that increment a variable, write it to a data file and read it back in
*****************************************/

#include "udf.h"

int kount = 0;  /* define global variable kount */
At the top of the listing, the integer kount is defined and initialized to zero. The first function (demo_calc) is an ADJUST function that increments the value of kount at each iteration, since the ADJUST function is called once per iteration. (See Section 4.2.1 for more information about ADJUST functions.) The second function (writer) instructs FLUENT to write the current value of kount to the data file, when the data file is saved. The third function (reader) instructs FLUENT to read the value of kount from the data file, when the data file is read.

The functions work together as follows. If you run your calculation for, say, 10 iterations (kount has been incremented to a value of 10) and save the data file, then the current value of kount (10) will be written to your data file. If you read the data back into FLUENT and continue the calculation, kount will start at a value of 10 and will be incremented at each iteration. Note that you can save as many static variables as you want, but you must be sure to read them in the same order in which they are written.

**Hooking a Read/Write Case or Data File UDF to FLUENT**

After the UDF that you have defined using DEFINE_RW_FILE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.1.5 for details on how to hook your DEFINE_RW_FILE UDF to FLUENT.
4.3 Model-Specific DEFINE Macros

The DEFINE macros presented in this section are used to set parameters for a particular model in FLUENT. Table 4.3.1 provides a quick reference guide to the DEFINE macros, the functions they are used to define, and the panels where they are activated in FLUENT. Definitions of each DEFINE macro are listed in the udf.h header file. For your convenience, they are listed in Appendix A.

- DEFINE_CHEM_STEP (Section 4.3.1)
- DEFINE_DELTAT (Section 4.3.2)
- DEFINE_DIFFUSIVITY (Section 4.3.3)
- DEFINE_DOM_DIFFUSE_REFLECTIVITY (Section 4.3.4)
- DEFINE_DOM_SOURCE (Section 4.3.5)
- DEFINE_DOM_SPECULAR_REFLECTIVITY (Section 4.3.6)
- DEFINE_HEAT_FLUX (Section 4.3.7)
- DEFINE_NET_REACTION_RATE (Section 4.3.8)
- DEFINE_NOX_RATE (Section 4.3.9)
- DEFINE_PR_RATE (Section 4.3.10)
- DEFINE_PRANDTL (Section 4.3.11)
- DEFINE_PROFILE (Section 4.3.12)
- DEFINE_PROPERTY (Section 4.3.13)
- DEFINE_SCAT_PHASE_FUNC (Section 4.3.14)
- DEFINE_SOURCE (Section 4.3.15)
- DEFINE_SR_RATE (Section 4.3.16)
- DEFINE_TURB_PREMIX_SOURCE (Section 4.3.17)
- DEFINE_TURBULENT_VISCOSITY (Section 4.3.18)
- DEFINE_UDS_FLUX (Section 4.3.19)
- DEFINE_UDS_UNSTEADY (Section 4.3.20)
- DEFINE_VR_RATE (Section 4.3.21)
## Table 4.3.1: Quick Reference Guide for Model-Specific DEFINE Macros

<table>
<thead>
<tr>
<th>Function</th>
<th>DEFINE Macro</th>
<th>Panel Activated In</th>
</tr>
</thead>
<tbody>
<tr>
<td>species mass fraction</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition (e.g., Velocity Inlet)</td>
</tr>
<tr>
<td>volume fraction</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>(multiphase models)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>velocity at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>pressure at a boundary</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>boundary temperature</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>turbulent kinetic energy</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>turbulent dissipation rate</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>wall thermal conditions</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>wall shear stress</td>
<td>DEFINE_PROFILE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>mass source</td>
<td>DEFINE_SOURCE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>momentum source</td>
<td>DEFINE_SOURCE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>energy source</td>
<td>DEFINE_SOURCE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>turbulent k.e. source</td>
<td>DEFINE_SOURCE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>turb. dissipation rate source</td>
<td>DEFINE_SOURCE</td>
<td>boundary condition</td>
</tr>
<tr>
<td>heat flux</td>
<td>DEFINE_HEAT_FLUX</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>turbulent viscosity</td>
<td>DEFINE_TURBULENT_VISCOITY</td>
<td>Viscous Model</td>
</tr>
<tr>
<td>Function</td>
<td>DEFINE Macro</td>
<td>Panel Activated In</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>----------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>density</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>viscosity</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>mass diffusivity</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>thermal conductivity</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>absorption coefficient</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>scattering coefficient</td>
<td>DEFINEPROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>laminar flow speed</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>rate of strain</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>species or UDS diffusivity</td>
<td>DEFINE_DIFFUSIVITY</td>
<td>Materials</td>
</tr>
<tr>
<td>scattering phase function</td>
<td>DEFINE_SCAT_PHASE_FUNC</td>
<td>Materials</td>
</tr>
<tr>
<td>surface reaction rate</td>
<td>DEFINE_SR_RATE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>volume reaction rate</td>
<td>DEFINE_VR_RATE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>particle reaction rate</td>
<td>DEFINE_PR_RATE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>species net mass reaction rate</td>
<td>DEFINE_NETREACTION_RATE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>species net mass reaction rate</td>
<td>DEFINE_CHEM_STEP</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>integrated over time step</td>
<td></td>
<td></td>
</tr>
<tr>
<td>turbulent premixed source</td>
<td>DEFINE_TURB_PREMIX_SOURCE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>scalar flux function</td>
<td>DEFINE_UDS_FLUX</td>
<td>User-Defined Scalars</td>
</tr>
<tr>
<td>scalar unsteady function</td>
<td>DEFINE_UDS_UNSTEADY</td>
<td>User-Defined Scalars</td>
</tr>
<tr>
<td>Function</td>
<td>DEFINE Macro</td>
<td>Panel Activated In</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>NO\textsubscript{x} formation rate</td>
<td>DEFINE_NOX_RATE</td>
<td>NO\textsubscript{x} Model</td>
</tr>
<tr>
<td>time step (for time dependent solutions)</td>
<td>DEFINE_DELTAT</td>
<td>Iterate</td>
</tr>
<tr>
<td>diffusive reflectivity (for discrete ordinates (DO) model)</td>
<td>DEFINE_DOM_DIFFUSE_REFLECTIVITY</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>specular reflectivity (for DO model)</td>
<td>DEFINE_DOM_SPECULAR_REFLECTIVITY</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>source (for DO model)</td>
<td>DEFINE_DOM_SOURCE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>Prandtl numbers</td>
<td>DEFINE_PRANDTL</td>
<td>Viscous Model</td>
</tr>
<tr>
<td>user-defined mixing law (density viscosity, conductivity) for mixture materials</td>
<td>DEFINE_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>surface tension coefficient for VOF multiphase model</td>
<td>DEFINE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
</tbody>
</table>
4.3 Model-Specific DEFINE Macros

4.3.1 DEFINE_CHEM_STEP

Description

You can use the DEFINE_CHEM_STEP macro to return the net mass reaction rate of all species, integrated over the provided time step:

\[ Y_i^{\Delta t} = Y_i^0 + \int_0^{\Delta t} \frac{dY_i}{dt} dt \]  \hspace{1cm} (4.3-1)

where \( Y_i^0 \) is the initial mass fraction of species \( i \), \( t \) is time, \( \Delta t \) is the given time step, and \( \frac{dY_i}{dt} \) is the net mass reaction rate. \( Y_i^{\Delta t} \) is \( i \)'th species mass fraction at the end of the integration.

DEFINE_CHEM_STEP UDFs are used for the EDC and PDF Transport models.

Usage

Macro: DEFINE_CHEM_STEP (name, ifail, n, dt, p, temp, yk)

Argument Types: int *ifail  
int n  
real dt  
real *p  
real *temp  
real *yk

Function returns: void

There are seven arguments to DEFINE_CHEM_STEP: name, ifail, n, dt, p, temp, and yk. You will supply name, the name of the UDF. ifail, n, dt, p, temp, and yk are variables that are passed by the FLUENT solver to your UDF.

The passed argument, ifail, is a flag that is set to 0 if the integration was successful, and otherwise returns a non-zero value. n is the number of volumetric species. dt is the time step, p is pressure, temp is temperature, and yk is an array containing the initial species mass fractions. The output of the function is the array of mass fractions yk after the integration step. The initial mass fractions in array yk are overwritten.
DEFINE Macros

Example

The following UDF, named user_chem_step, assumes that the net volumetric reaction rate is the expression,

\[
\frac{dY_k}{dt} = \frac{1}{N_{spe}} - Y_k
\]  

(4.3-2)

where \( N_{spe} \) is the number of species.

An analytic solution exists for the integral of this ODE as,

\[
Y_k^{\Delta t} = (Y_k^0 - 1/N_{spe}) \exp(-\Delta t) + 1/N_{spe}
\]  

(4.3-3)

/****************************

Example UDF that demonstrates DEFINE_CHEM_STEP

******************************************************************************
#include "udf.h"

DEFINE_CHEM_STEP(user_chem_step, ifail, n, dt, p, temp, yk)
{
    int i;
    real c = 1./n_spe, decay = exp(-dt);
    for(i=0;i<n_spe;i++)
        yk[i] = (yk[i]-c)*decay + c;
}

Hooking a Chemistry Step UDF to FLUENT

After the UDF that you have defined using DEFINE_CHEM_STEP is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.1 for details on how to hook your DEFINE_CHEM_STEP UDF to FLUENT.
4.3 Model-Specific DEFINE Macros

4.3.2 DEFINE_DELTAT

Description

You can use the DEFINE_DELTAT macro to control the size of the time step during the solution of a time-dependent problem. Note that this macro can be used only if the adaptive time-stepping method option has been activated in the Iterate panel in FLUENT.

Usage

Macro: DEFINE_DELTAT (name, domain)

Argument types: Domain *domain

Function returns: real

There are two arguments to DEFINE_DELTAT: name and domain. You will supply name, the name of the UDF. domain is passed by the FLUENT solver to your UDF. Your UDF will need to compute the real value of the physical time step and return it to the solver.

domain is a pointer to the domain over which the time-stepping control function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the domain pointer that is passed to the function by the solver is the mixture-level domain pointer.

Example

The following UDF, named mydeltat, is a simple function that shows how you can use DEFINE_DELTAT to change the value of the time step in a simulation. First, CURRENT_TIME is used to get the value of the current simulation time (which is assigned to the variable flow_time). Then, for the first 0.5 seconds of the calculation, a time step of 0.1 is set. A time step of 0.2 is set for the remainder of the simulation. The time step variable is then returned to the solver. See Section 6.7 for details on CURRENT_TIME.
DEFINE Macros

UDF that changes the time step value for a time-dependent solution
******************************************************************************/
#include "udf.h"

DEFINE_DELTAT(mydeltat, domain)
{
    real time_step;
    real flow_time = CURRENT_TIME;
    if (flow_time < 0.5)
        time_step = 0.1;
    else
        time_step = 0.2;
    return time_step;
}

Hooking an Adaptive Time Step UDF to FLUENT

After the UDF that you have defined using DEFINE_DELTAT is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Iterate panel in FLUENT. See Section 8.2.2 for details on how to hook your DEFINE_DELTAT UDF to FLUENT.
4.3.3 DEFINE_DIFFUSIVITY

Description

You can use the DEFINE_DIFFUSIVITY macro to specify the diffusivity for the species transport equations or user-defined scalar (UDS) transport equations.

Usage

Macro:       DEFINE_DIFFUSIVITY (name, c, t, i)

Argument types: 

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>cell_t</td>
</tr>
<tr>
<td>c</td>
<td>int</td>
</tr>
<tr>
<td>t</td>
<td>Thread *t</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
</tr>
</tbody>
</table>

Function returns: real

There are four arguments to DEFINE_DIFFUSIVITY: name, c, and t, and i. You will supply name, the name of the UDF. c, t, and i are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to compute the diffusivity only for a single cell and return the real value to the solver.

c is an index that identifies a cell within the given thread. t is a pointer to the thread on which the diffusivity function is to be applied. i is an index that identifies the species or user-defined scalar.

Note that diffusivity UDFs are called by FLUENT from within a loop on cell threads. Consequently, your UDF will not need to loop over cells in a thread since FLUENT is doing it outside of the function call.
DEFINE Macros

Example

The following UDF, named mean_age_diff, computes the diffusivity for the mean age of air using a user-defined scalar. Note that the mean age of air calculations do not require that energy, radiation, or species transport calculations have been performed. You will need to set uds-0 = 0.0 at all inlets and outlets in your model. This function can be executed as an interpreted or compiled UDF.

```c
#include "udf.h"

DEFINE_DIFFUSIVITY(mean_age_diff, c, t, i)
{
    return C_R(c, t) * 2.88e-05 + C_MU_EFF(c, t) / 0.7;
}
```

Hooking a Diffusivity UDF to FLUENT

After the UDF that you have defined using DEFINE_DIFFUSIVITY is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Materials panel in FLUENT. See Section 8.2.3 for details on how to hook your DEFINE_DIFFUSIVITY UDF to FLUENT.
4.3 Model-Specific DEFINE Macros

4.3.4 DEFINE_DOM_DIFFUSE_REFLECTIVITY

Description

You can use the DEFINE_DOM_DIFFUSE_REFLECTIVITY macro when you want to modify the inter-facial reflectivity computed by FLUENT at diffusely reflecting semi-transparent walls, based on the refractive index values. During execution, a DEFINE_DOM_DIFFUSE_REFLECTIVITY function is called by FLUENT for each semi-transparent wall and also for each band (in the case of a non-gray Discrete Ordinates Model). Therefore the function can be used to modify diffuse reflectivity and diffuse transmissivity values at the interface.

Usage

Macro: DEFINE_DOM_DIFFUSE_REFLECTIVITY (name, t, nb, n_a, n_b, diff_ref_a, diff_tran_a, diff_ref_b, diff_tran_b)

Argument types:
Thread *t
int nb
real n_a
real n_b
real *diff_ref_a
real *diff_tran_a
real *diff_ref_b
real *diff_tran_b

Function returns: void

There are nine arguments to DEFINE_DOM_DIFFUSE_REFLECTIVITY: name, t, nb, n_a, n_b, diff_ref_a, diff_tran_a, diff_ref_b, and diff_tran_b. You will supply name, the name of the UDF. t, nb, n_a, n_b, diff_ref_a, diff_tran_a, diff_ref_b, and diff_tran_b are variables that are passed by the FLUENT solver to your UDF.

The passed variable, t, is a pointer to the thread on which the discrete ordinate diffusivity function is to be applied. The band number is represented by nb. This argument is needed for the non-gray Discrete Ordinates Model (DOM). n_a and n_b represent the refractive indexes of medium a and b. The variables diff_ref_a and diff_tran_a are the diffuse reflectivity and diffuse transmissivity, respectively, at the interface facing medium a. Similarly, diff_ref_b and diff_tran_b are the diffuse reflectivity and diffuse transmissivity, respectively, at the interface facing medium b.
DEFINE Macros

Example

The following UDF, named \texttt{user\_dom\_diff\_refl}, modifies diffuse reflectivity and transmissivity values on both the sides of the interface separating medium \textit{a} and \textit{b}. The UDF is called for all the semi-transparent walls and prints the value of the diffuse reflectivity and transmissivity values for side \textit{a} and \textit{b}.

/* UDF to print the diffuse reflectivity and transmissivity at semi-transparent walls*/
#include "udf.h"

DEFINE_DOM_DIFFUSE_REFLECTIVITY(user_dom_diff_refl, t, nband, n_a, n_b, 
  diff_ref_a, diff_tran_a, diff_ref_b, diff_tran_b)
{
  /* Argument types
   Thread *t
   int nband
   int n_a
   int n_b
   real *diff_ref_a // diffuse reflectivity on side a
   real *diff_tran_a // diffuse transmissivity on side a
   real *diff_ref_b // diffuse reflectivity on side b
   real *diff_tran_b // diffuse transmissivity on side b */

  printf("diff\_ref\_a=%f diff\_tran\_a=%f \n", *diff_ref_a, *diff_tran_a);
  printf("diff\_ref\_b=%f diff\_tran\_b=%f \n", *diff_ref_b, *diff_tran_b);
}

Hooking a Discrete Ordinates Model (DOM) Diffuse Reflectivity UDF to FLUENT

After the UDF that you have defined using \texttt{DEFINE\_DOM\_DIFFUSE\_REFLECTIVITY} is interpreted or compiled (see Chapter 7 for details), the name that you specified in the \texttt{DEFINE} macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.4 for details on how to hook your \texttt{DEFINE\_DOM\_DIFFUSE\_REFLECTIVITY} UDF to FLUENT.
4.3 Model-Specific DEFINE Macros

4.3.5 DEFINE_DOM_SOURCE

Description

You can use the DEFINE_DOM_SOURCE macro to alter the emission term (first term on the right hand side of equation 11.3-37 or 11.3-38 in the User’s Guide) as well as the scattering term (second term on the right hand side of equation 11.3-37 or 11.3-38) in the radiative transport equation for the Discrete Ordinates (DO) model.

Usage

Macro:        DEFINE_DOM_SOURCE (name, c, t, ni, nb, emission,
               in_scattering, abs_coeff, scat_coeff)

Argument types:  cell_t c
                   Thread *t
                   int ni
                   int nb
                   real *emission
                   real *in_scattering
                   real *abs_coeff
                   real *scat_coeff

Function returns:  void

There are nine arguments to DEFINE_DOM_SOURCE: name, c, ni, nb, emission, in_scattering, abs_coeff, and scat_coeff. You will supply name, the name of the UDF. c, ni, nb, emission, in_scattering, abs_coeff, and scat_coeff are variables that are passed by the FLUENT solver to your UDF.

c is an index that identifies the cell. t is a pointer to the cell thread. ni denotes the direction represented by the solid angle. nb is the band number. This argument is needed for the non-gray DO model. emission and in_scattering are pointers to the emission and scattering term in the radiative transport equation 11.3-37 or 11.3-38 in the User’s Guide), respectively. abs_coeff and scat_coeff are pointers to absorption and scattering coefficients, respectively.

DEFINE_DOM_SOURCE is called by FLUENT for each cell.
Example

In the following UDF, named `user_dom_source`, the emission term present in the radiative transport equation is modified. The UDF is called for all the cells and increases the emission term by 5%.

```c
/* UDF to alter the emission source term in the DO model */

#include "udf.h"

DEFINE_DOM_SOURCE(user_dom_source, c, t, ni, nb, emission, in_scattering, abs_coeff, scat_coeff)
{
    /* increased the emission by 5 % */

    *emission *= 1.05;

}
```

Hooking a DOM Source UDF to FLUENT

After the UDF that you have defined using `DEFINE_DOM_SOURCE` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.5 for details on how to hook your `DEFINE_DOM_SOURCE` UDF to FLUENT.
4.3.6 DEFINE_DOM_SPECULAR_REFLECTIVITY

**Description**

You can use the `DEFINE_DOM_SPECULAR_REFLECTIVITY` macro to alter the inter-facial reflectivity of specularly reflecting semi-transparent walls. You may wish to do this if the reflectivity is dependent on other conditions that the standard boundary condition doesn’t allow for. (See Specular Semi-Transparent Walls in the User’s Guide for more details.) During FLUENT execution, the same UDF is called for all the faces of the semi-transparent wall, for each of the directions.

**Usage**

Macro:  
`DEFINE_DOM_SPECULAR_REFLECTIVITY (name, f, t, nband, n_a, n_b, ray_direction, en, internal_reflection,  
specular_reflectivity, specular_transmissivity)`

Argument types:  
- `face_t f`  
- `Thread *t`  
- `int nband`  
- `real n_a`  
- `real n_b`  
- `real ray_direction`  
- `real en`  
- `int internal_reflection`  
- `real *specular_reflectivity`  
- `real *specular_transmissivity`

Function returns:  
`void`

There are eleven arguments to `DEFINE_DOM_SPECULAR_REFLECTIVITY`: `name`, `f`, `t`, `nband`, `n_a`, `n_b`, `ray_direction`, `en`, `internal_reflection`, `specular_reflectivity`, and `specular_transmissivity`. You will supply `name`, the name of the UDF. `f`, `t`, `nband`, `n_a`, `n_b`, `ray_direction`, `en`, `internal_reflection`, `specular_reflectivity`, and `specular_transmissivity` are variables that are passed by the FLUENT solver to your UDF.

The passed variable, `f`, is an index that identifies a face and `t` is a pointer to the face thread on which the specular reflectivity function is to be applied. The band number is represented by `nband` which is needed for the non-gray Discrete Ordinates (DO) model. `n_a` and `n_b` represent the refractive indexes of medium a and b, respectively. `ray_direction` is the direction vector (s in Equation 11.3.50 in the User’s Guide). `en` is the interface normal (n in Equation 11.3.50). `internal_reflection` is used to
flag the code that total internal reflector has occurred. \texttt{specular\_reflectivity} and \texttt{specular\_transmissivity} are the specular reflectivity and specular transmissivity, respectively, for the given direction \textit{s}.

**Example**

In the following UDF, named \texttt{user\_dom\_spec\_refl}, specular reflectivity and transmissivity values are altered for a given ray direction \textit{s} at face \textit{f}.

```c
/* UDF to alter the specular reflectivity and transmissivity, at semi-transparent walls, along direction \textit{s} at face \textit{f} */
#include "udf.h"
DEFINE_DOM_SPECULAR_REFLECTIVITY(user_dom_spec_refl, f, t, nband, n_a, n_b, ray_direction, en, internal\_reflection, specular\_reflectivity, specular\_transmissivity)
{
    /* Argument types
     * face\_t f
     * Thread *t
     * int nband
     * int n\_a (refractive index of medium a)
     * int n\_b (refractive index of medium b)
     * real ray\_direction (\textit{s} : direction vector defined in eqn 11.3.50)
     * real en (\textit{n} : normal vector defined in eqn 11.3.50 in the manual)
     * int *internal\_reflection (1 : total internal reflection)
     * real *specular\_reflectivity (specular reflectivity on side a)
     * real *specular\_transmissivity {specular transmissivity on side a}
     */
    real angle, cos\_theta;
    real PI = 3.141592;
    cos\_theta = NV\_DOT(ray\_direction, en);
    angle = acos(cos\_theta);
    if (angle > 45 && angle < 60)
    {
        *specular\_reflectivity = 0.3;
        *specular\_transmissivity = 0.7;
    }
}
```
Hooking a Discrete Ordinate Model (DOM) Specular Reflectivity UDF to FLUENT

After the UDF that you have defined using \texttt{DEFINE\_DOM\_SPECULAR\_REFLECTIVITY} is interpreted or compiled (see Chapter 7 for details), the name that you specified in the \texttt{DEFINE} macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.6 for details on how to hook your \texttt{DEFINE\_DOM\_SPECULAR\_REFLECTIVITY} UDF to FLUENT.
4.3.7 DEFINE_HEAT_FLUX

Description

In spite of its name, the DEFINE_HEAT_FLUX macro is not to be used to explicitly set the heat flux along a wall. FLUENT computes the heat flux along a wall based on currently selected models to account for the diffusive and radiative energy fluxes (if any). You must only use a DEFINE_HEAT_FLUX UDF when you want to employ some other heat transfer mechanism that is not currently being modeled. The total heat flux at the wall will be the sum of the currently computed heat flux (based on the activated models) and the heat flux defined by the UDF.

Usage

Macro: DEFINE_HEAT_FLUX (name, f, t, c0, t0, cid, cir)

Argument types: face_t f
Thread *t
cell_t c0
Thread *t0
real cid[]
real cir[]

Function returns: void

There are seven arguments to DEFINE_HEAT_FLUX: name, f, t, c0, t0, cid, and cir. You will supply name, the name of the UDF. f, t, c0, t0, cir[], and cid[] are variables that are passed by the FLUENT solver to your UDF.

The passed variable, f, is an index that identifies a wall face within the given thread. t is a pointer to the thread on which the heat flux function is to be applied. c0 is an index that identifies the cell next to the wall, and t0 is a pointer to the adjacent cell’s thread.

cid[] and cir[] are real arrays that need to be computed by your UDF. Array cid[] stores the fluid-side diffusive heat transfer coefficients, while array cir[] stores radiative heat transfer coefficients. With these inputs provided to the function, the diffusive heat flux (qid) and radiative heat flux (qir) are computed by FLUENT according to the following equations:

\[
qid = cid[0] + cid[1]*C_T(c0,t0) - cid[2]*F_T(f,t) - cid[3]*pow(F_T(f,t),4)
\]
\[
qir = cir[0] + cir[1]*C_T(c0,t0) - cir[2]*F_T(f,t) - cir[3]*pow(F_T(f,t),4)
\]

The sum of qid and qir defines the total heat flux from the fluid to the wall (this direction being positive flux), and, from an energy balance at the wall, equals the heat
flux of the surroundings (exterior to the domain). Note that heat flux UDFs (defined using `DEFINE_HEAT_FLUX`) are called by FLUENT from within a loop over wall faces.

In order for the solver to compute $C_T$ and $F_T$, the values you supply to `cid[1]` and `cid[2]` should never be zero.

**Example**

Section 11.5.2 provides an example of the P-1 radiation model implementation through a user-defined scalar. An example of the usage of the `DEFINE_HEAT_FLUX` macro is included in that implementation.

**Hooking a Heat Flux UDF to FLUENT**

After the UDF that you have defined using `DEFINE_HEAT_FLUX` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.7 for details on how to hook your `DEFINE_HEAT_FLUX` UDF to FLUENT.
4.3.8 DEFINE_NET_REACTION_RATE

Description

You can use the DEFINE_NET_REACTION_RATE macro to return the net mass reaction rate of all species. The net mass reaction rate of a species is the sum over all reactions of the volumetric reaction rate, multiplied by the molecular weight of that species:

\[
\frac{dY_i}{dt} = M_i \sum_{r=1}^{N_R} \hat{R}_r
\]

(4.3-4)

where \(Y_i\) is the mass fraction of species \(i\), \(t\) is time, \(M_i\) is the molecular weight of species \(i\), \(\hat{R}_r\) is the molar reaction rate of reaction \(r\), and \(N_R\) is the total number of reactions.

A DEFINE_NET_REACTION_RATE UDF may be used for the EDC and PDF Transport models, as well as for the surface chemistry model. In contrast, the volumetric UDF function DEFINE_VR_RATE and surface UDF function DEFINE_SR_RATE return the molar rate per reaction. You might want to use DEFINE_NET_REACTION_RATE to link to an external program that provides reaction rates, such as CHEMKIN.

Usage

Macro: 
DEFINE_NET_REACTION_RATE (name, p, temp, yi, rr, jac)

Argument Types: 
double *p
double *temp
double *yi
double *rr
double *jac

Function returns: void

There are six arguments to DEFINE_NET_REACTION_RATE: name, p, temp, yi, rr, and jac. You will supply name, the name of the UDF. p, temp, yi, rr, and jac are variables that are passed by the FLUENT solver to your UDF.

The input variables are pressure p, temperature temp, and the array of species mass fractions yi. The outputs of the function are the array of net mass reaction rates, rr, and the Jacobian array jac. Note that the Jacobian is only required for surface chemistry, and is the derivative of the surface net mass reaction rate with respect to the species concentration.
Example

The following UDF, named user_net_reaction_rate, assumes that the net volumetric reaction rate is the expression,

\[
\frac{dY_i}{dt} = \frac{1}{N_{spe}} - Y_i
\]  

(4.3-5)

where \(N_{spe}\) is the number of species.

```c
#include "udf.h"

DEFINE_NET_REACTION_RATE(user_net_reaction_rate, p, temp, yi, rr, jac)
{
    int i;
    for(i=0;i<n_spe;i++)
        rr[i] = 1./(real)n_spe - yi[i];
}
```

Hooking a Net Mass Reaction Rate UDF to FLUENT

After the UDF that you have defined using DEFINE_NET_REACTION_RATE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.8 for details on how to hook your DEFINE_NET_REACTION_RATE UDF to FLUENT.
4.3.9  DEFINE NOX_RATE

Description

You can use the DEFINE_NOX_RATE macro to calculate NO\textsubscript{x} production and reduction rates in FLUENT. The UDF rate that you specify is independent of the standard NO\textsubscript{x} model options. You can deselect the standard NO\textsubscript{x} options in your simulation, and choose the UDF rate instead.

Usage

Macro:  

\texttt{DEFINE\_NOX\_RATE (name, c, t, NOx)}

Argument types:  

\begin{itemize}
  \item \texttt{cell\_t c}
  \item \texttt{Thread *t}
  \item \texttt{NOx\_Parameter *NOx}
\end{itemize}

Function returns:  \texttt{void}

There are four arguments to \texttt{DEFINE\_NOX\_RATE}: \texttt{name}, \texttt{c}, \texttt{t}, and \texttt{NOx}. You will supply \texttt{name}, the name of the UDF. \texttt{c}, \texttt{t}, and \texttt{NOx} are variables that are passed by the FLUENT solver to your UDF.

\texttt{c} is an index that identifies a cell within the given thread. \texttt{t} is a pointer to the thread on which the NO\textsubscript{x} rate is to be applied. \texttt{NOx} is a pointer to the NO\textsubscript{x} structure. A \texttt{DEFINE\_NOX\_RATE} function does not return a value. The calculated NO\textsubscript{x} rates are returned through the NO\textsubscript{x} structure.

Note that although the data structure is called NO\textsubscript{x}, the \texttt{DEFINE\_NOX\_RATE} macro can be used to calculate the rates of any of the pollutant species (i.e., NO, HCN, and NH\textsubscript{3}), depending on which of the pollutant species equations is being solved.
Example

The following compiled UDF, named `user_nox`, computes NO\textsubscript{x} production and reduction rates based on the forward and reverse rates of NO defined as

\[ R_f = \frac{2[O]k_1k_2[O_2][N_2]}{k_2[O_2] + k_{-1}[NO]} \] (4.3-6)

and

\[ R_r = \frac{2[O]k_{-1}k_{-2}[NO]^2}{k_2[O_2] + k_{-1}[NO]} \] (4.3-7)

where the rate coefficients, which have units of m\textsuperscript{3}/mol-s, are defined as

\[ k_1 = 1.8 \times 10^8 \exp\left(\frac{-38370}{T}\right) \] (4.3-8)

\[ k_{-1} = 3.8 \times 10^7 \exp\left(\frac{-425}{T}\right) \] (4.3-9)

\[ k_2 = 1.8 \times 10^4T \exp\left(\frac{-4680}{T}\right) \] (4.3-10)

\[ k_{-2} = 3.8 \times 10^4T \exp\left(\frac{-20820}{T}\right) \] (4.3-11)

O concentration is given by

\[ [O] = 3.664 \times 10^1T^{0.5}[O_2]^{0.5}\exp\left(\frac{-27123}{T}\right) \] (4.3-12)

All concentrations in the rate expressions have units of mol/m\textsuperscript{3}.

/* *************************************************************************/
UDF example of User-Defined NOx Rate
/* *************************************************************************/

#include "udf.h"

#define SMALL_S 1.e-29
DEFINE Macros

DEFINE_NOX_RATE(user_nox, c, t, NOx)
{
    real kf1, kr1, kf2, kr2;
    real o_eq;
    real s1, s2, s3, rf, rr;

    Rate_Const K_F[2] = {{1.80e8, 0.0, 38370.0},
                        {1.80e4, 1.0, 4680.0}};

    Rate_Const K_R[2] = {{3.80e7, 0.0, 425.0},
                        {3.80e3, 1.0, 20820.0}};

    Rate_Const K_O = {3.664e1, 0.5, 27123.0};

    if (NOX_EQN(NOx) != EQ_NO) return;

    kf1 = ARRH(NOx, K_F[0]);
    kr1 = ARRH(NOx, K_R[0]);
    kf2 = ARRH(NOx, K_F[1]);
    kr2 = ARRH(NOx, K_R[1]);

    s1 = kf2*MOLECON(NOx, O2);
    s3 = s1 + kr1*MOLECON(NOx, NO);

    /* determine O concentration (partial equilibrium)*/
    o_eq = ARRH(NOx, K_O)*sqrt(MOLECON(NOx, O2));

    /* calculate NO rate */
    s2 = 2.*o_eq;
    /* forward rate... */
    rf = s2*(kf1*MOLECON(NOx, N2))*s1/s3;
    /* reverse rate... */
    rr = -s2*kr1*kr2*pow(MOLECON(NOx, NO), 2.0)/s3;

    /* rates have units mole/m^3/s */
    NOX_FRATE(NOx) = rf;
    NOX_RRATE(NOx) = rr;
}

A number of Fluent-supplied macros can be used in the calculation of pollutant rate using UDFs. The macros listed below are defined in the header file sg_nox.h, which is included in the udf.h file. The variable NOx indicates a pointer to the NOx_Parameter structure.
4.3 Model-Specific DEFINE Macros

- **NOX_EQN(NOx)** returns the index of the pollutant equation currently being solved. The indices’ are EQ_NO for NO, EQ_HCN for HCN, and EQ_NH3 for NH3.

- **MOLECON(NOx, SPE)** returns the molar concentration of a species specified by SPE, which must be replaced by one of the following identifiers: FUEL, O2, O, OH, H2O, N2, N, CH, CH2, CH3, NO, HCN, NH3. Identifier FUEL represents the fuel species as specified in the Fuel Species drop-down list under Prompt NO Parameters in the NOx Model panel.

- **NULLIDX(NOx, SPE)** returns TRUE if the species specified by SPE does not exist in the FLUENT case (i.e., in the Species panel).

- **ARRH(NOx, K)** returns the Arrhenius rate calculated from the constants specified by K. K is defined using the Rate_Const data type and has three elements - A, B, and C. The Arrhenius rate is given in the form of

\[ R = AT^B \exp(-C/T) \]

where \( T \) is the temperature.

- **NOX_FRATE(NOx)** is used to return the production rate of the pollutant species being solved.

- **NOX_RRATE(NOx)** is used to return the reduction rate of the pollutant species being solved.

**Hooking a NOx Rate UDF to FLUENT**

After the UDF that you have defined using DEFINE_NOX_RATE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the NOx Model panel in FLUENT. See Section 8.2.9 for details on how to hook your DEFINE_NOX_RATE UDF to FLUENT.
4.3.10 DEFINE_PR_RATE

Description

You can use the DEFINE_PR_RATE macro when you want to define a custom particle surface reaction for the multiple surface reactions particle model. During FLUENT execution, the same UDF is called sequentially for all particle surface reactions, so DEFINE_PR_RATE can be used to define custom reaction rates for a single reaction, or for multiple reactions. The volumetric and wall surface reactions are not affected by the definition of this macro and will follow the designated rates. Note that a DEFINE_PR_RATE UDF is not called with the coupled solution option, so you will need to disable the Coupled Heat Mass Solution option in the Discrete Phase Model panel when using it. The auxiliary function, zbrent_pr_rate, which is provided below, can be used when there is no analytical solution for the overall particle reaction rate.

Usage

Macro:

```
DEFINE_PR_RATE (name, c, t, r, mw, ci, p, sf, dif_index, 
cat_index, rr)
```

Argument Types:

- cell_t c
- Thread *t
- Reaction *r
- real *mw
- real *ci
- Tracked_Particle *p
- real *sf
- int dif_index
- int cat_index
- real *rr

Function returns: void

There are eleven arguments to DEFINE_PR_RATE: name, c, t, r, mw, ci, p, sf, dif_index, cat_index, and rr. You will supply name, the name of the UDF. c, t, r, mw, ci, p, sf, dif_index, cat_index, and rr are variables that are passed by the FLUENT solver to your UDF.

- c is an index that identifies the cell that the particle currently contains, and t is a pointer to the thread for the particle. r is a pointer to the data structure that represents the current reaction. mw is a real pointer array of the gaseous and surface species molecular weights. pp is a real pointer array of the gas partial pressures. p is a pointer to the Tracked_Particle data structure that contains data related to the particle being
tracked. This pointer can be used as an argument to the particle-specific macros defined in Section 5.6 to obtain information about particle properties. \( sf \) is a real pointer array containing the mass fractions of the solid species in the particle char mass at the current time step. Note that the order in which the solid species mass fractions are stored in array \( sf \) is the same as the order in which the species are defined in the Selected Solid Species list in the Materials panel, which is opened from the Edit Species names option for the Mixture Material. \( \text{dif}_i \) and \( \text{cat}_i \) are integer variables representing the diffusion controlled species and catalyst species, respectively, as defined in the Reactions panel for the current reaction. Your UDF will need to set the value referenced by the real pointer \( rr \) to the rate of particle reaction in kgmol/s.

**DEFINE PR RATE** is called by FLUENT every time step during the particle tracking calculation. The auxiliary function \( \text{zbrent.pr_rate} \) is used when there is no analytical solution for the overall particle reaction rate. It uses Brent’s method to find the root of a function known to lie between \( x_1 \) and \( x_2 \). The root will be refined until its accuracy has reached tolerance \( \text{tol} \). This is demonstrated in Example 2.

**Auxiliary function**

\[ \text{zbrent.pr_rate} \left(\text{real } (*\text{func}), (\text{real}, \text{real []}, \text{int []}, \text{boolean []}, \text{char *}), \text{real ruser[]}, \text{int iuser[]}, \text{boolean buser[]}, \text{char } *\text{cuser}, \text{real x1 real x2, real tol, boolean } *\text{ifail}\right) \]

Auxiliary function returns: real

**Example 1**

The following UDF, named \( \text{user.pr_rate} \), specifies a particle reaction rate given by equation 13.3-9 of the User’s Guide, where the effectiveness factor \( \eta_r \) is defined as

\[ \eta_r = 1 - x \]

where \( x \) is the fractional conversion of the particle char mass. In this case, the UDF will be applied to all surface particle reactions defined in the FLUENT model.
/* UDF of specifying the surface reaction rate of a particle */

#include "udf.h"

#define A1 0.002
#define E1 7.9e7

DEFINE_PR_RATE(user_pr_rate,c,t,r,mw,pp,p,sf,dif_i,cat_i,rr)
{
/* Argument types
    cell_t c
    Thread *t
    Reaction *r (reaction structure)
    real *mw (species molecular weight)
    real *pp (gas partial pressures)
    Tracked_Particle *p (particle structure)
    real *sf (current mass fractions of solid species in
      particle char mass)
    int dif_i (index of diffusion controlled species)
    int cat_i (index of catalyst species)
    real *rr (rate of reaction kgmol/s)
*/

real ash_mass =
P_INIT_MASS(p)*(1.-DPM_CHAR_FRACTION(p)-DPM_VOLATILE_FRACTION(p));

real one_minus_conv =
MAX(0.,(P_MASS(p) -ash_mass) / P_INIT_MASS(p)/ DPM_CHAR_FRACTION(p));

real rate = A1*exp(-E1/UNIVERSAL_GAS_CONSTANT/P_T(p));

*rr=-rate*P_DIAM(p)*P_DIAM(p)*M_PI*sf[0]*one_minus_conv;
}
Example 2

The following compiled UDF, named user_rate, specifies a particle reaction rate given by equations 13.3-4 to 13.3-7 of the User’s Guide. The reaction order on the kinetic rate is 0.9 and the effectiveness factor $\eta_r$ is defined as

$$
\eta_r = 1 - x
$$

where $x$ is the fractional conversion of the particle char mass. In this case it is necessary to obtain a numerical solution for the overall surface reaction rate.

This UDF is called only for reaction 2, which means that the default FLUENT solution will be used for the rest of the particle surface reactions defined.

```c
/* UDF of specifying the surface reaction rate of a particle, using a numerical solution */

#include "udf.h"

#define c1 5e-12
#define A1 0.002
#define E1 7.9e7
#define tolerance 1e-4
#define order 0.9

real reaction_rate(real rate, real ruser[], int iuser[], boolean buser[], char *cuser )
{
   return (ruser[2]*pow(MAX(0.,(ruser[0]-rate/ruser[1])),order) -rate);
}

DEFINE_PR_RATE(user_rate,c,t,r,mw,pp,p,sf,dif_i,cat_i,rr)
{
if (!strcmp(r->name, "reaction-2"))
{
   boolean ifail=FALSE;

   real ash_mass =
P_INIT_MASS(p)*(1.-DPM_CHAR_FRACTION(p)-DPM_VOLATILE_FRACTION(p));

   real one_minus_conv =
MAX(0.,(P_MASS(p) -ash_mass) / P_INIT_MASS(p)/ DPM_CHAR_FRACTION(p));
```
DEFINE Macros

```c
real ruser[3];
int iuser[1];
boolean buser[1];
char cuser[30];

real ratemin, ratemax, root;

ruser[0] = pp[dif_i];
ruser[1] = MAX(1.E-15, (c1*pow(0.5*(P_T(p)+C_T(c,t)),0.75)/P_DIAM(p)));
strcpy(cuser, "reaction-2");

ratemin=0;
ratemax=ruser[1]*pp[dif_i];

/* arguments for auxiliary function zbrent_pr_rate

In this example, a real function named reaction_rate is defined at the top of the UDF. The arguments of reaction_rate are real rate, and the pointer arrays real ruser[], integer iuser[], boolean buser[], and char *cuser, which must be declared and defined in the main body of the DEFINE_PR_RATE function.

Typically, if the particle surface reaction rate is described by

\[ rate = f(ruser[],iuser[],rate) \]

then the real function (in this example reaction_rate) should return

\[ f(ruser[],iuser[],rate) - rate \]

The variables boolean buser[] and char *cuser can be used to control the flow of the program in cases of complicated rate definitions.

ratemin and ratemax, hold the minimum and maximum possible values of the variable rate, respectively. They define the search interval where the numerical algorithm will search for the root of the equation, as defined in the function reaction_rate. The value of reaction rate \( \text{rr} \) will be refined until an accuracy specified by the value of tolerance \( \text{tol} \) is reached.

The variable ifail will take the value TRUE if the root of the function has not been found.
```
4.3 Model-Specific DEFINE Macros

```c
root = zbrent_pr_rate(reaction_rate, ruser, iuser, buser, cuser,
    ratemin, ratemax, tolerance, &ifail);

if (ifail) root=MAX(1.E-15,ruser[1]);

*rr=-root*P_DIAM(p)*P_DIAM(p)*M_PI*sf[0]*one_minus_conv;

Message("Fail status %d\n", ifail);
Message("Reaction rate for reaction %s : %g\n", cuser, *rr);
```

**Hooking a Particle Reaction Rate UDF to FLUENT**

After the UDF that you have defined using `DEFINE_PR_RATE` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.10 for details on how to hook your `DEFINE_PR_RATE` UDF to FLUENT.
4.3.11 DEFINE_PRANDTL

The following DEFINE macros can be used to specify Prandtl numbers in FLUENT.

DEFINE_PRANDTL_D

Description

You can use the DEFINE_PRANDTL_D macro to specify Prandtl numbers for turbulent dissipation ($\epsilon$).

Usage

Macro: DEFINE_PRANDTL_D (name, c, t)

Argument Types: cell_t c
Thread *t

Function returns: real

There are three arguments to DEFINE_PRANDTL_D: name, c, and t. You will supply name, the name of the UDF. c and t are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value for the turbulent dissipation Prandtl number to the solver.

The passed variable t is a pointer to the cell thread and c is an index that identifies the cell on which the Prandtl number function is to be applied.

Hooking a Prandtl Number UDF to FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_D is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Viscous Model panel in FLUENT. See Section 8.2.11 for details on how to hook your DEFINE_PRANDTL_D UDF to FLUENT.
DEFINE_PRANDTL_K

Description
You can use the DEFINE_PRANDTL_K macro to specify Prandtl numbers for turbulence kinetic energy \( (k) \).

Usage

Macro: DEFINE_PRANDTL_K (name, c, t)

Argument Types: cell_t c
Thread *t

Function returns: real

There are three arguments to DEFINE_PRANDTL_K: name, c, and t. You will supply name, the name of the UDF. c and t are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value for the kinetic energy Prandtl number to the solver.

The passed variable t is a pointer to the cell thread and c is an index that identifies the cell on which the Prandtl number function is to be applied.

Example
The following UDF implements a high-Re version of the RNG model, using the \( k-\epsilon \) option that is activated in FLUENT.

Three steps are required:

1. Set Cmu, C1eps, and C2eps as in the RNG model.
2. Calculate Prandtl numbers for \( k \) and \( \epsilon \) using the UDF.
3. Add the -r source term in the \( \epsilon \) equation.

In the RNG model, diffusion in \( k \) and \( \epsilon \) equations appears as

\[
(\mu + \mu_t) \alpha
\]
while in the standard $k$-$\epsilon$ model, it is given by

$$\mu + \frac{\mu_t}{Pr}.$$ .

For the new implementation, a UDF is needed to define a Prandtl number $Pr$ as

$$Pr = \frac{\mu_t}{[(\mu + \mu_t) \alpha - \mu]}$$

in order to achieve the same implementation as the original RNG Model.

The following functions (which are concatenated into a single C source code file) demonstrate this usage. Note that the source code must be executed as a compiled UDF.

```c
#include "udf.h"

DEFINE_PRANDTL_K(user_pr_k, c, t)
{
    real pr_k, alpha;
    real mu = C_MU_L(c,t);
    real mu_t = C_MU_T(c,t);

    alpha = rng_alpha(1., mu + mu_t, mu);

    pr_k = mu_t/((mu+mu_t)*alpha-mu);

    return pr_k;
}

DEFINE_PRANDTL_D(user_pr_d, c, t)
{
    real pr_d, alpha;
    real mu = C_MU_L(c,t);
    real mu_t = C_MU_T(c,t);

    alpha = rng_alpha(1., mu + mu_t, mu);

    pr_d = mu_t/((mu+mu_t)*alpha-mu);

    return pr_d;
}
```
4.3 Model-Specific DEFINE Macros

```c
DEFINE_SOURCE(eps_r_source, c, t, dS, eqn)
{
    real con, source;
    real mu   = C_MU_L(c,t);
    real mu_t = C_MU_T(c,t);
    real k    = C_K(c,t);
    real d    = C_D(c,t);
    real prod = C_PRODUCTION(c,t);

    real s = sqrt( prod/(mu+mu_t) ) ;
    real eta = s*k/d;
    real eta_0 = 4.38;
    real term = mu_t*s*s*s/(1.0 + 0.012*eta*eta*eta);

    source = - term * (1. - eta/eta_0);
    dS[eqn] = - term/d;

    return source;
}
```

**Hooking a Prandtl Number UDF to FLUENT**

After the UDF that you have defined using `DEFINE_PRANDTL_K` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Viscous Model panel in FLUENT. See Section 8.2.11 for details on how to hook your `DEFINE_PRANDTL_K` UDF to FLUENT.
DEFINE Macros

DEFINE_PRANDTL_O

Description

You can use the DEFINE_PRANDTL_O macro to specify Prandtl numbers for specific dissipation ($\omega$ in the $k$-$\omega$ model).

Usage

Macro: DEFINE_PRANDTL_O (name, c, t)

Argument Types: cell_t c
Thread *t

Function returns: real

There are three arguments to DEFINE_PRANDTL_O: name, c, and t. You will supply name, the name of the UDF. c and t are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value for the specific dissipation Prandtl number to the solver.

The passed variable t is a pointer to the cell thread and c is an index that identifies the cell on which the Prandtl number function is to be applied.

Example

/* Specifying a Constant Specific Dissipation Prandtl Number */

#include "udf.h"

DEFINE_PRANDTL_O(user_pr_o, c, t)
{
    real pr_o;
    pr_o = 2.;
    return pr_o;
}

Hooking a Prandtl Number UDF to FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_O is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Viscous Model panel in FLUENT. See Section 8.2.11 for details on how to hook your DEFINE_PRANDTL_O UDF to FLUENT.


4.3 Model-Specific DEFINE Macros

DEFINE_PRANDTL_T

Description

You can use the DEFINE_PRANDTL_T macro to specify Prandtl numbers that appear in the temperature equation diffusion term.

Usage

Macro: DEFINE_PRANDTL_T (name, c, t)

Argument Types: cell_t c
Thread *t

Function returns: real

There are three arguments to DEFINE_PRANDTL_T: name, c, and t. You will supply name, the name of the UDF. c and t are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value for the temperature Prandtl number to the solver.

The passed variable t is a pointer to the cell thread and c is an index that identifies the cell on which the Prandtl number function is to be applied.

Example

/* Specifying a Constant Temperature Prandtl Number */

#include "udf.h"

DEFINE_PRANDTL_T(user_pr_t, c, t)
{
    real pr_t;
    pr_t = 0.85;
    return pr_t;
}

Hooking a Prandtl Number UDF to FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_T is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Viscous Model panel in FLUENT. See Section 8.2.11 for details on how to hook your DEFINE_PRANDTL_T UDF to FLUENT.
DEFINE Macros

DEFINE_PRANDTL_T_WALL

Description
You can use the DEFINE_PRANDTL_T_WALL macro to specify Prandtl numbers for thermal wall functions.

Usage

Macro: DEFINE_PRANDTL_T_WALL (name, c, t)

Argument Types: cell_t c
Thread *t

Function returns: real

There are three arguments to DEFINE_PRANDTL_T_WALL: name, c, and t. You will supply name, the name of the UDF. c and t are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value for the thermal wall function Prandtl number to the solver.

The passed variable t is a pointer to the cell thread and c is an index that identifies the cell on which the Prandtl number function is to be applied.

Example

/********************************************
 Specifying a constant thermal wall function Prandtl number
 *********************************************/
DEFINE_PRANDTL_T_WALL(user_pr_t_wall, c, t)
{
    real pr_t_wall;
    pr_t_wall = 0.85;
    return pr_t_wall;
}

Hooking a Prandtl Number UDF to FLUENT

After the UDF that you have defined using DEFINE_PRANDTL_T_WALL is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Viscous Model panel in FLUENT. See Section 8.2.11 for details on how to hook your DEFINE_PRANDTL_T_WALL UDF to FLUENT.
4.3.12 DEFINE_PROFILE

Description

You can use the DEFINE_PROFILE macro to define a custom boundary profile that varies as a function of spatial coordinates or time. Some of the variables you can customize at a boundary are:

- velocity, pressure, temperature, turbulent kinetic energy, turbulent dissipation rate
- species mass fraction (species transport)
- volume fraction (multiphase)
- wall thermal conditions (temperature, heat flux, heat generation rate, heat transfer coefficients, and external emissivity, etc.)
- wall shear and stress conditions
- etc.

Note that DEFINE_PROFILE allows you to modify only a single value for wall heat flux. Single values are used in the explicit source term which FLUENT does not linearize. If you want to linearize your source term for wall heat flux and account for conductive and radiative heat transfer separately, you will need to use DEFINE_HEAT_FLUX to specify your UDF.

Some examples of boundary profile UDFs are provided below.

Usage

Macro: DEFINE_PROFILE (name, t, i)

Argument types: Thread *t
int i

Function returns: void

There are three arguments to DEFINE_PROFILE: name, t, and i. You will supply name, the name of the UDF. t and i are variables that are passed by the FLUENT solver to your UDF.

The passed variable t is a pointer to the thread on which the boundary condition is to be applied. i is an index that identifies the variable that is to be defined. i is set when
you hook the UDF with a variable in a boundary condition panel through the graphical user interface. This index is subsequently passed to your UDF by the FLUENT solver, so that your function knows which variable to operate on.

While DEFINE_PROFILE is usually used to specify a profile condition on a boundary face zone, it can also be used to specify, or fix, flow variables that are held constant during computation in a cell zone. (See Section 6.26 of the User’s Guide for more information on fixing values in a cell zone boundary condition.) For these cases, the arguments of the macro will change accordingly.

Note that unlike source term and property UDFs, profile UDFs (defined using DEFINE_PROFILE) are not called by FLUENT from within a loop on threads in the boundary zone. The solver passes only the pointer to the thread associated with the boundary zone to the DEFINE_PROFILE macro. Your UDF will need to do the work of looping over all of the faces in the thread, computing the face value for the boundary variable, and then storing the value in memory. Fluent has provided you with a face looping macro to loop over all faces in a thread (begin_f_loop...). See Chapter 6 for details about face looping macro utilities.

F_PROFILE is typically used along with DEFINE_PROFILE and is a predefined macro supplied by Fluent. F_PROFILE stores a boundary condition in memory for a given face and thread and is nested within the face loop as shown in the examples below. It is important to note that the index $i$ that is an argument to DEFINE_PROFILE is the same argument to F_PROFILE. F_PROFILE uses the thread pointer $t$, face identifier $f$, and index $i$ to set the appropriate boundary face value in memory. See Section 6.4 for a description of F_PROFILE.

In multiphase cases a DEFINE_PROFILE UDF may be called more than once (particularly if the profile is used in a mixture domain thread). If this needs to be avoided, then add the prefix MP_ to the UDF name. The function will then be called only once even if it is used for more than one profile.
Example 1 - Pressure Profile

The following UDF, named `pressure_profile`, generates a parabolic pressure profile according to the equation

\[ p(y) = 1.1 \times 10^5 - 0.1 \times 10^5 \left( \frac{y}{0.0745} \right)^2 \]

Note that this UDF assumes that the grid is generated such that the origin is at the geometric center of the boundary zone to which the UDF is to be applied. \( y \) is 0.0 at the center of the inlet and extends to \( \pm 0.0745 \) at the top and bottom of the inlet. The source code can be interpreted or compiled in FLUENT.

```
#include "udf.h"

DEFINE_PROFILE(pressure_profile, t, i)
{
    real x[ND_ND];  /* this will hold the position vector */
    real y;
    face_t f;

    begin_f_loop(f, t)
    {
        F_CENTROID(x,f,t);
        y = x[1];
        F_PROFILE(f, t, i) = 1.1e5 - y*y/(.0745*.0745)*0.1e5;
    }
    end_f_loop(f, t)
}
```

The function named `pressure_profile` has two arguments: \( t \) and \( i \). \( t \) is a pointer to the face's thread, and \( i \) is an integer that is a numerical label for the variable being set within each loop.

Within the function body variable \( f \) is declared as a face. A one-dimensional array \( x \) and variable \( y \) are declared as `real` data types. Following the variable declarations, a looping macro is used to loop over each face in the zone to create a profile, or an array of data. Within each loop, `F_CENTROID` returns the value of the face centroid (array \( x \)) for the face with index \( f \) that is on the thread pointed to by \( t \). The \( y \) coordinate stored
in \( x[1] \) is assigned to variable \( y \), and is then used to calculate the pressure. This value is then assigned to \texttt{F PROFILE} which uses the integer \( i \) (passed to it by the solver, based on your selection of the UDF as the boundary condition for pressure in the \textit{Pressure Inlet} panel) to set the pressure face value in memory.

**Example 2 - Velocity, Turbulent Kinetic Energy, and Turbulent Dissipation Rate Profiles**

In the following example, \texttt{DEFINE PROFILE} is used to generate profiles for the \( x \) velocity, turbulent kinetic energy, and dissipation rate, respectively, for a 2D fully-developed duct flow. Three separate UDFs named \texttt{x velocity}, \texttt{k profile}, and \texttt{dissip profile} are defined. These functions are concatenated in a single C source file and can be interpreted or compiled in \texttt{FLUENT}.

The 1/7th power law is used to specify the \( x \) velocity component:

\[
v_x = v_{x,\text{free}} \left( \frac{y}{\delta} \right)^{1/7}
\]

A fully-developed profile occurs when \( \delta \) is one-half the duct height. In this example, the mean \( x \) velocity is prescribed and the peak (free-stream) velocity is determined by averaging across the channel.

The turbulent kinetic energy is assumed to vary linearly from a near-wall value of

\[
k_{nw} = \frac{u^2}{\sqrt{C_\mu}}
\]

to a free-stream value of

\[
k_{inf} = 0.002u^2_{\text{free}}
\]

The dissipation rate is given by

\[
\epsilon = \frac{C_{\mu}^{3/4}(k^{3/2})}{\ell}
\]

where the mixing length \( \ell \) is the minimum of \( \kappa y \) and 0.085\( \delta \). (\( \kappa \) is the von Karman constant = 0.41.)

The friction velocity and wall shear take the forms:

\[
u_r = \sqrt{\tau_w/\rho}
\]
\[ \tau_w = \frac{f \rho u_{\text{free}}^2}{2} \]

The friction factor is estimated from the Blasius equation:

\[ f = 0.045 \left( \frac{u_{\text{free}} \delta}{\nu} \right)^{-1/4} \]

/* Concatenated UDFs for fully-developed turbulent inlet profiles */

`#include "udf.h"`

`#define YMIN 0.0 /* constants */`
`#define YMAX 0.4064`
`#define UMEAN 1.0`
`#define B 1./7.`
`#define DELOVRH 0.5`
`#define VISC 1.7894e-05`
`#define CMU 0.09`
`#define VKC 0.41`

`/* profile for x-velocity */`

`DEFINE_PROFILE(x_velocity, t, i)`
{
    `real y, del, h, x[ND_ND], ufree; /* variable declarations */`
    `face_t f;`

    `h = YMAX - YMIN;`
    `del = DELOVRH*h;`
    `ufree = UMEAN*(B+1.);`

    `begin_f_loop(f, t)`
    `{`
        `F_CENTROID(x,f,t);`
        `y = x[1];`
DEFINE Macros

if (y <= del)
    F_PROFILE(f,t,i) = ufree*pow(y/del,B);
else
    F_PROFILE(f,t,i) = ufree*pow((h-y)/del,B);
} end_f_loop(f, t)

/* profile for kinetic energy */

DEFINE_PROFILE(k_profile, t, i)
{
    real y, del, h, ufree, x[ND_ND];
    real ff, utau, knw, kinf;
    face_t f;

    h = YMAX - YMIN;
    del = DELOVRH*h;
    ufree = UMEAN*(B+1.);
    ff = 0.045/pow(ufree*del/VISC,0.25);
    utau=sqrt(ff*pow(ufree,2.)/2.0);
    knw=pow(utau,2.)/sqrt(CMU);
    kinf=0.002*pow(ufree,2.);

    begin_f_loop(f, t)
    {
        F_CENTROID(x,f,t);
        y=x[1];

        if (y <= del)
            F_PROFILE(f,t,i)=knw+y/del*(kinf-knw);
        else
            F_PROFILE(f,t,i)=knw+(h-y)/del*(kinf-knw);
    }
    end_f_loop(f, t)
}
/* profile for dissipation rate */

DEFINE_PROFILE(dissip_profile, t, i)
{
    real y, x[ND_ND], del, h, ufree;
    real ff, utau, knw, kinf;
    real mix, kay;
    face_t f;

    h = YMAX - YMIN;
    del = DELOVRH*h;
    ufree = UMEAN*(B+1.);
    ff = 0.045/pow(ufree*del/VISC,0.25);
    utau=sqrt(ff*pow(ufree,2.)/2.0);
    knw=pow(utau,2.)/sqrt(CMU);
    kinf=0.002*pow(ufree,2.);

    begin_f_loop(f, t)
    {
        F_CENTROID(x,f,t);
        y=x[1];

        if (y <= del)
            kay=knw+y/del*(kinf-knw);
        else
            kay=knw+(h-y)/del*(kinf-knw);

        if (VKC*y < 0.085*del)
            mix = VKC*y;
        else
            mix = 0.085*del;

        F_PROFILE(f,t,i)=pow(CMU,0.75)*pow(kay,1.5)/mix;
    }
    end_f_loop(f,t)
}
Example 3 - Fixed Velocity UDF

In the following example DEFINE_PROFILE is used to fix flow variables that are held constant during computation in a cell zone. Three separate UDFs named fixed_u, fixed_v, and fixed_ke are defined in a single C source file. They specify fixed velocities that simulate the transient startup of an impeller in an impeller-driven mixing tank. The physical impeller is simulated by fixing the velocities and turbulence quantities using the fix option in FLUENT. See Section 6.26 of the User’s Guide for more information on fixing variables.

/*#include "udf.h"*/

#define FLUID_ID 1
#define ua1 -7.1357e-2
#define ua2 54.304
#define ua3 -3.1345e3
#define ua4 4.5578e4
#define qa1 2.2723e-2
#define ka2 6.7989
#define ka3 -424.18
#define ka4 9.4615e3
#define ka5 -7.7251e4
#define ka6 1.8410e5
#define da1 -6.5819e-2
#define da2 88.845
#define da3 -5.3731e3
#define da4 1.1643e5
#define da5 -9.1202e5
#define da6 1.9567e6
DEFINE_PROFILE(fixed_u, t, i)
{
    cell_t c;
    real x[ND_ND];
    real r;

    begin_c_loop (c,t)
    {
        /* centroid is defined to specify position dependent profiles*/
        C_CENTROID(x,c,t);
        r =x[1];
        F_PROFILE(c,t,i) = ua1+(ua2*r)+(ua3*r*r)+(ua4*r*r*r)+(ua5*r*r*r*r);
    }
    end_c_loop (c,t)
}

DEFINE_PROFILE(fixed_v, t, i)
{
    cell_t c;
    real x[ND_ND];
    real r;

    begin_c_loop (c,t)
    {
        /* centroid is defined to specify position dependent profiles*/
        C_CENTROID(x,c,t);
        r =x[1];
        F_PROFILE(c,t,i) = va1+(va2*r)+(va3*r*r)+(va4*r*r*r)+(va5*r*r*r*r);
    }
    end_c_loop (c,t)
}
DEFINE_PROFILE(fixed_ke, t, i)
{
    cell_t c;
    real x[ND_ND];
    real r;

    begin_c_loop (c,t)
    {
        /* centroid is defined to specify position dependent profiles*/
        C_CENTROID(x,c,t);
        r =x[1];
        F_PROFILE(c,t,i) =
        ka1+(ka2*r)+(ka3*r*r)+(ka4*r*r*r)+(ka5*r*r*r*r)+(ka6*r*r*r*r*r);
    }
    end_c_loop (c,t)
}

Example 4 - Wall Heat Generation Rate Profile

The following UDF, named wallheatgenerate, generates a heat generation rate profile for a planar conduction wall. Once interpreted or compiled, you can activate this UDF in the Wall boundary condition panel in FLUENT.

    /* Wall Heat Generation Rate Profile UDF */
    /*/include "udf.h"*/

DEFINE_PROFILE(wallheatgenerate,thread,i)
{
    real source = 0.001;
    face_t f;

    begin_f_loop(f,thread)
    {
        F_PROFILE(f,thread,i) = source;
    }
    end_f_loop(f,thread)
}

Hooking a Boundary Profile UDF to FLUENT

After the UDF that you have defined using DEFINE_PROFILE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the appropriate boundary condition panel (e.g., the Velocity Inlet panel) in FLUENT. See Section 8.2.12 for details on how to hook your DEFINE_PROFILE UDF to FLUENT.
4.3 Model-Specific DEFINE Macros

4.3.13 DEFINE_PROPERTY

Description

You can use the DEFINE_PROPERTY macro to specify a custom material property in FLUENT for single-phase and multiphase flows. When you are writing a user-defined mixing law UDF for a mixture material, you will need to use special utilities to access species material properties. These are described below. If you want to define a custom mass diffusivity property, you must use DEFINE_DIFFUSIVITY instead of DEFINE_PROPERTY. See Section 4.3.3.

Some of the properties you can customize using DEFINE_PROPERTY are:

- density (as a function of temperature only)
- viscosity
- thermal conductivity
- absorption and scattering coefficients
- laminar flow speed
- rate of strain
- particle or droplet diameter (multiphase mixture model)
- user-defined mixing laws for density, viscosity, and conductivity of mixture materials
- surface tension coefficient (multiphase VOF model)

UDFs cannot be used to define specific heat properties; specific heat data can only be accessed and not modified in FLUENT.

Usage

Macro:  DEFINE_PROPERTY (name, c, t)

Argument types:  cell_t c
                 Thread *t

Function returns:  real
There are three arguments to `DEFINE_PROPERTY`: `name`, `c`, and `t`. You will supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to compute the real property only for a single cell and return it to the solver. The passed variable `t` is a pointer to the cell thread and `c` is an index that identifies the cell on which the property function is to be applied.

Note that like source term UDFs, property UDFs (defined using `DEFINE_PROPERTY`) are called by FLUENT from within a loop on cell threads. The solver passes all of the variables needed to allow a `DEFINE_PROPERTY` UDF to define a custom material, since properties are assigned on a cell basis. Consequently, your UDF will not need to loop over cells in a zone since FLUENT is already doing it.

**Auxiliary Utilities**

Some commonly-used auxiliary utilities for custom property UDFs are described below. They are `generic_property`, `MATERIAL_PROPERTY`, `THREAD_MATERIAL`, and `mixture_species_loop`.

**Function:**

`generic_property(name, c, t, prop, id, T)`

**Argument types:**

- `cell_t c`
- `Thread *t`
- `Property *prop`
- `Property_ID id`
- `real T`

**Function returns:**

`real`

generic_property is a general purpose function that returns the real value for the given property id for the given thread material. It is defined in `prop.h` and is used only for species properties.

t is a pointer to the cell thread and c is an index that identifies the cell on which the property function is to be applied. prop is a pointer to the property array for the thread material and can be obtained through the macro `MATERIAL_PROPERTY(m)` (see below). T is the temperature at which the property is to be evaluated (used only if a polynomial method is specified). id is the Property_ID of the required property you want to define a custom mixing law for (e.g., `PROP_ktc` for thermal conductivity). The following Property_ID variables are available:

- `PROP_rho`, density
- `PROP_mu`, viscosity
• PROP_ktc, thermal conductivity

Macro: MATERIALPROPERTY (m)

Argument types: Material *m

MATERIALPROPERTY is defined in materials.h and returns a pointer to the Property array prop for the given material pointer m.

Macro: THREAD_MATERIAL (t)

Argument types: Thread *t

THREAD_MATERIAL is defined in threads.h and returns the pointer m to the Material that is associated with the given cell thread t.

! Note that in previous versions of FLUENT, THREAD_MATERIAL took two arguments (t,i), but now only takes one (t).

Macro: mixture_species_loop (m,sp,i)

Argument types: Material *m, Material *sp, int i

mixture_species_loop is defined in materials.h and loops over all of the species for the given mixture material.
Example 1 - Temperature-dependent Viscosity Property

The following UDF, named cell_viscosity, generates a variable viscosity profile to simulate solidification. The function is called for every cell in the zone. The viscosity in the warm ($T > 288$ K) fluid has a molecular value for the liquid ($5.5 \times 10^{-3}$ kg/m-s), while the viscosity for the cooler region ($T < 286$ K) has a much larger value (1.0 kg/m-s). In the intermediate temperature range (286 K $\leq T \leq$ 288 K), the viscosity follows a linear profile that extends between the two values given above:

$$\mu = 143.2135 - 0.49725T$$ \hspace{1cm} (4.3-13)

This model is based on the assumption that as the liquid cools and rapidly becomes more viscous, its velocity will decrease, thereby simulating solidification. Here, no correction is made for the energy field to include the latent heat of freezing. The source code can be interpreted or compiled in FLUENT.

/*******************************************************************************/
UDF that simulates solidification by specifying a temperature-dependent viscosity property
*******************************************************************************/

DEFINE_PROPERTY(cell_viscosity, c, t) {

real mu_lam;
real temp = C_T(c, t);

if (temp > 288.)
  mu_lam = 5.5e-3;
else if (temp > 286.)
  mu_lam = 143.2135 - 0.49725 * temp;
else
  mu_lam = 1.;

return mu_lam;
}

The function cell_viscosity is defined on a cell. Two real variables are introduced: temp, the value of C_T(c,t), and mu_lam, the laminar viscosity computed by the function. The value of the temperature is checked, and based upon the range into which it falls, the appropriate value of mu_lam is computed. At the end of the function the computed value for the viscosity (mu_lam) is returned to the solver.
Example 2 - User-defined Mixing Law for Thermal Conductivity

You can use DEFINE_PROPERTY to define custom user-defined mixing laws for density, viscosity, and conductivity of mixture materials. In order to access species material properties your UDF will need to utilize auxiliary utilities that are described above.

The following UDF, named mass_wtd_k, is an example of a mass-fraction weighted conductivity function. The UDF utilizes the generic_property function to obtain properties of individual species. It also makes use of MATERIALPROPERTY and THREAD_MATERIAL.

/******************************************************************************
 UDF that specifies a custom mass-fraction weighted conductivity
******************************************************************************/

DEFINE_PROPERTY(mass_wtd_k,c,t)
{
    real sum = 0.; int i;
    Material *sp;
    real ktc;
    Property *prop;
    mixture_species_loop(THREAD_MATERIAL(t),sp,i)
    {
        prop = (MATERIALPROPERTY(sp));
        ktc = generic_property(c,t,prop,PROP_ktc,C_T(c,t));
        sum += C_YI(c,t,i)*ktc;
    }
    return sum;
}
Example 3 - Surface Tension Coefficient UDF

DEFINE_PROPERTY can also be used to define a surface tension coefficient UDF for the multiphase VOF model. The following UDF specifies a surface tension coefficient as a quadratic function of temperature. The source code can be interpreted or compiled in FLUENT.

/***************************************************************
Surface Tension Coefficient UDF for the VOF Multiphase Model
***************************************************************/

#include "udf.h"
DEFINE_PROPERTY(sfc, c, t)
{
    real T = C_T(c, t);
    return 1.35 - 0.004*T + 5.0e-6*T*T;
}

Hooking a Property UDF to FLUENT

After the UDF that you have defined using DEFINE_PROPERTY is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Materials panel in FLUENT. See Section 8.2.13 for details on how to hook your DEFINE_PROPERTY UDF to FLUENT.
4.3.14 DEFINE_SCAT_PHASE_FUNC

Description

You can use the DEFINE_SCAT_PHASE_FUNC macro to define the radiation scattering phase function for the Discrete Ordinates (DO) model. The function computes two values: the fraction of radiation energy scattered from direction $i$ to direction $j$, and the forward scattering factor.

Usage

Macro: DEFINE_SCAT_PHASE_FUNC (name, cosine, f)

Argument types: real cosine
                real *f

Function returns: real

There are three arguments to DEFINE_SCAT_PHASE_FUNC: name, cosine, and f. You will supply name, the name of the UDF. cosine and f are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to compute the real fraction of radiation energy scattered from direction $i$ to direction $j$ and return it to the solver.

The passed variable cosine is a real variable that is the cosine of the angle between the two directions. f is a real pointer that points to the location in memory where the real forward scattering factor is stored. Note that the solver computes and stores a scattering matrix for each material by calling this function for each unique pair of discrete ordinates.
DEFINE Macros

Example

In the following example, a number of UDFs are concatenated in a single C source file. These UDFs implement backward and forward scattering phase functions that are cited by Jendoubi et al. [1]. The source code can be interpreted or compiled in FLUENT.

/************************************************************
UDFs that implement backward and forward scattering
phase functions as cited by Jendoubi et. al.
************************************************************/

#include "udf.h"

DEFINE_SCAT_PHASE_FUNC(ScatPhiB2, c, fsf)
{
  real phi=0;
  *fsf = 0;
  phi = 1.0 - 1.2*c + 0.25*(3*c*c-1);
  return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiB1, c, fsf)
{
  real phi=0;
  *fsf = 0;
  phi = 1.0 - 0.56524*c + 0.29783*0.5*(3*c*c-1) +
  0.08571*0.5*(5*c*c*c-3*c) + 0.01003/8*(35*c*c*c-30*c*c+3) +
  0.00063/8*(63*c*c*c*c-70*c*c*c+15*c);
  return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiF3, c, fsf)
{
  real phi=0;
  *fsf = 0;
  phi = 1.0 + 1.2*c + 0.25*(3*c*c-1);
  return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiF2, c, fsf)
{
  real phi=0;
  real coeffs[9]={1,2,0.00917,1.56339,0.67407,0.22215,0.04725,
  0.00671,0.00068,0.00005};
real P[9];
int i;
*fsf = 0;
P[0] = 1;
P[1] = c;
phi = P[0]*coeffs[0] + P[1]*coeffs[1];
for(i=1;i<7;i++)
{
    P[i+1] = 1/(i+1.0)*((2*i+1)*c*P[i] - i*P[i-1]);
    phi += coeffs[i+1]*P[i+1];
}
return (phi);

DEFINE_SCAT_PHASEFUNC(ScatIso, c, fsf)
{
    *fsf=0;
    return (1.0);
}

**Hooking a DEFINE_SCAT_PHASE_FUNC UDF to FLUENT**

After the UDF that you have defined using `DEFINE_SCAT_PHASE_FUNC` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the Materials panel in FLUENT. See Section 8.2.13 for details on how to hook your `DEFINE_SCAT_PHASE_FUNC` UDF to FLUENT.
4.3.15 DEFINE_SOURCE

Description

You can use the DEFINE_SOURCE macro to specify custom source terms for the different types of solved transport equations in FLUENT (except the discrete ordinates radiation model) including:

- continuity
- momentum
- $k$, $\epsilon$
- energy (also for solid zones)
- species mass fractions
- user-defined scalar (UDS) transport

Usage

Macro: DEFINE_SOURCE (name, c, t, dS, eqn)

Argument types:

- cell_t c
- Thread *t
- real dS[
- int eqn

Function returns: real

There are five arguments to DEFINE_SOURCE: name, c, t, dS, and eqn. You will supply name, the name of the UDF. c, t, dS, and eqn are variables that are passed by the FLUENT solver to your UDF.

The passed variable t is a pointer to the cell thread and c is an index that identifies the cell on which the source term is to be applied. Array dS specifies the derivative of the source term with respect to the dependent variable of the transport equation. These derivatives may be used to linearize the source term if they enhance the stability of the solver. To see this, note that the source term can be expressed, in general, as Equation 4.3-14, where $\phi$ is the dependent variable, $A$ is the explicit part of the source term, and $B\phi$ is the implicit part.

$$S_\phi = A + B\phi \quad (4.3-14)$$
Specifying a value for $B$ in Equation 4.3-14 can enhance the stability of the solution and help convergence rates due to the increase in diagonal terms on the solution matrix. \texttt{FLUENT} automatically determines if the value of $B$ that is given by the user will aid stability. If it does, then \texttt{FLUENT} will define $A$ as $S^* - \left(\frac{\partial S}{\partial \phi}\right)^* \phi^*$, and $B$ as $\left(\frac{\partial S}{\partial \phi}\right)^*$. If not, the source term is handled explicitly.

Your UDF will need to compute the \texttt{real} source term \textit{only} for a single cell and return the value to the solver, but you have the choice of setting the implicit term $dS[\texttt{eqn}]$ to $dS/d\phi$, or forcing the explicit solution of the source term by setting it equal to 0.

Note that like property UDFs, source term UDFs (defined using \texttt{DEFINE\_SOURCE}) are called by \texttt{FLUENT} from within a loop on cell threads. The solver passes to the \texttt{DEFINE\_SOURCE} term UDF all the necessary variables it needs to define a custom source term, since source terms are solved on a cell basis. Consequently, your UDF will \textit{not} need to loop over cells in the thread since \texttt{FLUENT} is already doing it.

The units on all source terms are of the form generation-rate/volume. For example, a source term for the continuity equation would have units of kg/m$^3$-s.
Example

The following UDF, named \texttt{xmom\_source}, is used to add source terms in \textsc{Fluent}. The source code can be interpreted or compiled. The function generates an $x$-momentum source term that varies with $y$ position as

$$
\text{source} = -0.5C_2\rho y|v_x|v_x
$$

Suppose

$$
\text{source} = S = -A|v_x|v_x
$$

where

$$
A = 0.5C_2\rho y
$$

Then

$$
\frac{dS}{dv_x} = -A|v_x| - Av_x \frac{d}{dv_x} (|v_x|)
$$

The source term returned is

$$
\text{source} = -A|v_x|v_x
$$

and the derivative of the source term with respect to $v_x$ (true for both positive and negative values of $v_x$) is

$$
\frac{dS}{dv_x} = -2A|v_x|
$$
4.3 Model-Specific DEFINE Macros

#include "udf.h"

#define C2 100.0

DEFINE_SOURCE(xmom_source, c, t, dS, eqn)
{
    real x[ND_ND];
    real con, source;

    C_CENTROID(x, c, t);
    con = C2*0.5*C_R(c, t)*x[1];

    source = -con*fabs(C_U(c, t))*C_U(c, t);
    dS[eqn] = -2.*con*fabs(C_U(c, t));

    return source;
}

Hooking a Source UDF to FLUENT

After the UDF that you have defined using DEFINE_SOURCE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Fluid or Solid boundary condition panel in FLUENT. See Section 8.2.15 for details on how to hook your DEFINE_SOURCE UDF to FLUENT.
4.3.16 DEFINE_SR_RATE

Description

You can use the DEFINE_SR_RATE macro when you want to define a custom surface reaction rate. A custom surface reaction rate function defined using this macro will overwrite the default reaction rate (e.g., finite-rate) that is specified in the Materials panel. An example of a reaction rate that depends upon gas species mass fractions is provided below. Also provided is a reaction rate UDF that takes into account site species.

Usage

Macro: DEFINE_SR_RATE (name, f, t, r, my, yi, rr)

Argument types: face_t f
Thread *t
Reaction *r
real *mw
real *yi
real *rr

Function returns: void

There are seven arguments to DEFINE_SR_RATE: name, f, t, r, my, yi, and rr. You will supply name, the name of the UDF. Once your UDF is compiled and linked, the name that you have chosen for your function will become visible and selectable in the graphical user interface in FLUENT. f, t, r, my, and yi are variables that are passed by the FLUENT solver to your UDF.

The passed variable t is a pointer to the thread on which the surface rate reaction is to be applied. f is an index that identifies a face within the given thread (or cell in the case of surface reaction in a porous zone). r is a pointer to the data structure for the reaction. mw is a pointer to a real array containing the species molecular weights, and yi is a pointer to a real array containing mass fractions of gas species at the surface and the coverage of site species (or site fractions).

Your UDF will need to set the reaction rate to the value referenced by the real pointer rr as shown in the examples below.
Example 1 - Surface Reaction Rate Using Species Mass Fractions

The following compiled UDF, named `arrhenius`, defines a custom surface reaction rate using species mass fractions in FLUENT.

```c
#include "udf.h"

/* ARRHENIUS CONSTANTS */
#define PRE_EXP 1e+15
#define ACTIVE 1e+08
#define BETA 0.0

real arrhenius_rate(real temp)
{
    return
    PRE_EXP*pow(temp,BETA)*exp(-ACTIVE/(UNIVERSAL_GAS_CONSTANT*temp));
}

/* Species numbers. Must match order in Fluent panel */
#define HF 0
#define WF6 1
#define H2O 2
#define NUM_SPECS 3

/* Reaction Exponents */
#define HF_EXP 2.0
#define WF6_EXP 0.0
#define H2O_EXP 0.0
#define MW_H2 2.0
#define STOIC_H2 3.0

/* Reaction Rate Routine that is used in both UDFs */
real reaction_rate(cell_t c, Thread *cthread,real mw[],real yi[])
{
    real concenHF = C_R(c, cthread)*yi[HF]/mw[HF];

    return arrhenius_rate(C_T(c,
```
DEFINE Macros

cthread)*pow(concenHF,HF_EXP);
}

DEFINE_SR_RATE(arrhenius,f,fthread,r,mw,yi,rr)
{
*rr =
reaction_rate(F_C0(f,fthread),THREAD_TO(f,fthread),mw,yi);
}

real contact_area(cell_t c, Thread *t, int s_id, int *n)
{
int i = 0;
real area = 0.0, A[ND_ND];

*n = 0;
c_face_loop(c,t,i)
{
if(THREAD_ID(C_FACE_THREAD(c,t,i)) == s_id)
{
(*n)++;
F_AREA(A, C_FACE(c,t,i), C_FACE_THREAD(c,t,i));
area += NV_MAG(A);
}
}
return area;
}

Example 2 - Surface Reaction Rate for Site Species

The following compiled UDF, named my_rate, defines a custom surface reaction rate that takes into account site species.

/****************************************************************************
 Custom surface reaction rate UDF
****************************************************************************/

#include "udf.h"

/* ARRHENIUS CONSTANTS */
#define PRE_EXP 1e+15
#define ACTIVE 1e+08
#define BETA 0.0
real arrhenius_rate(real temp)
{
    return
PRE_EXP*pow(temp,BETA)*exp(-ACTIVE/(UNIVERSAL_GAS_CONSTANT*temp));
}

/* Species numbers. Must match order in Fluent panel */
define HF 0
#define WF6 1
#define H2O 2
#define NUM_SPECS 3

/* Reaction Exponents */
define HF_EXP 2.0
#define WF6_EXP 0.0
#define H2O_EXP 0.0

#define MW_H2 2.0
#define STOIC_H2 3.0

/* Reaction Rate Routine that is used in both UDFs */
real reaction_rate(cell_t c, Thread *cthread,real mw[],real yi[])
{
    real concenHF = C_R(c, cthread)*yi[HF]/mw[HF];

    return arrhenius_rate(C_T(c, cthread))*pow(concenHF,HF_EXP);
}

DEFINE_SR_RATE(arrhenius,f,fthread,r,mw,yi,rr)
{
    *rr = reaction_rate(F_C0(f,fthread),THREAD_T0(f,fthread),mw,yi);
}

real contact_area(cell_t c, Thread *t, int s_id, int *n)
{
    int i = 0;
    real area = 0.0, A[ND_ND];

    *n = 0;
c_face_loop(c,t,i)
{
    if(THREAD_ID(C_FACE_THREAD(c,t,i)) == s_id)
    {
        (*n)++;
        F_AREA(A, C_FACE(c,t,i), C_FACE_THREAD(c,t,i));
        area += NV_MAG(A);
    }
}
return area;

/*******************************
Custom surface reaction rate UDF
******************************/
/* #include "udf.h" */

DEFINE_SR_RATE(my_rate, f, t, r, mw, yi, rr)
{
    Thread *t0=t->t0;
    cell_t c0=F_C0(f,t);
    real sih4 = yi[0]; /* mass fraction of sih4 at the wall */
    real si2h6 = yi[1];
    real sih2  = yi[2];
    real h2    = yi[3];
    real ar    = yi[4]; /* mass fraction of ar at the wall */

    real rho_w = 1.0, site_rho = 1.0e-6, T_w = 300.0;
    real si_s  = yi[6]; /* site fraction of si_s*/
    real sih_s = yi[7]; /* site fraction of sih_s*/
    T_w = F_T(f,t);
    rho_w = C_R(c0,t0)*C_T(c0,t0)/T_w;

    sih4  *= rho_w/mw[0]; /* converting of mass fractions
to molar concentrations */
    si2h6 *= rho_w/mw[1];
    sih2  *= rho_w/mw[2];
    h2    *= rho_w/mw[3];
    ar    *= rho_w/mw[4];
si_s *= site_rho;    /* converting of site fractions to site concentrations */
sih_s *= site_rho;

if (STREQ(r->name, "reaction-1"))
    *rr = 100.0*sih4;
else if (STREQ(r->name, "reaction-2"))
    *rr = 0.1*sih_s;
else if (STREQ(r->name, "reaction-3"))
    *rr = 100*si2h6*si_s;
else if (STREQ(r->name, "reaction-4"))
    *rr = 1.0e10*sih2;
}

Hooking a Surface Reaction Rate UDF to FLUENT

After the UDF that you have defined using DEFINE_SR_RATE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.16 for details on how to hook your DEFINE_SR_RATE UDF to FLUENT.
4.3.17 DEFINE_TURB_PREMIX_SOURCE

Description

You can use the DEFINE_TURB_PREMIX_SOURCE macro to customize the turbulent flame speed and source term in the premixed combustion model (see Chapter 15 of the User’s Guide) and the partially premixed combustion model (see Chapter 16).

Usage

Macro:          DEFINE_TURB_PREMIX_SOURCE (name, c, t, turb_flame_speed, source)

Argument types: cell_t c
                Thread *t
                real *turb_flame_speed
                real *source

Function returns: void

There are five arguments to DEFINE_TURB_PREMIX_SOURCE: name, c, t, turb_flame_speed, and source. You will supply name, the name of the UDF. c, t, turb_flame_speed, and source are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to set the turbulent flame speed to the value referenced by the turb_flame_speed pointer. It will also need to set the source term to the value referenced by the source pointer.

The passed variable t is a pointer to the thread on which the turbulent premixed source term is to be applied. c is an index that identifies a cell within the given thread. turb_flame_speed is a real pointer to the turbulent flame speed and source is a real pointer to the reaction progress source term.

Example

The following UDF, named turb_flame_src, specifies a custom turbulent flame speed and source term in the premixed combustion model. The source code must be executed as a compiled UDF in FLUENT.

In the standard premixed combustion model in FLUENT, the mean reaction rate of the progress variable (that is, the source term) is modeled as

\[ \rho S_c = \rho u U_t |\nabla c| \]  \hspace{1cm} (4.3-15)
where \( c \) is the mean reaction progress variable, \( \rho \) is the density, and \( U_t \) is the turbulent flame speed.

In the UDF example, the turbulent flame speed is modeled as

\[
U_t = U_l \sqrt{1 + (u'/U_l)^2}
\]

(4.3-16)

where \( U_l \) is the laminar flame speed and \( u' \) is the turbulent fluctuation. Note that the partially premixed combustion model is assumed to be enabled (see Chapter 16 of the User’s Guide), so that the unburned density and laminar flame speed are available as polynomials. See Chapter 6 for details on the NULLP, THREAD_STORAGE, and SV_VARS utilities.
/*********************************************************************************************/
UDF that specifies a custom turbulent flame speed and source
for the premixed combustion model
***********************************************************************************************/

#include "udf.h"
#include "sg_pdf.h" /* not included in udf.h so must include here */

DEFINE_TURB_PREMIX_SOURCE(turb_flame_src, c, t, turb_flame_speed, source)
{
    real up = TRB_VEL_SCAL(c,t);
    real ut, ul, grad_c, rho_u, DV[ND_ND];

    ul   = Ul_par_premix(C_FMEAN(c,t));
    rho_u = rho_par_premix(C_FMEAN(c,t));

    if( NNULLP(THREAD_STORAGE(t,SV_PREMIXC_G)) )
    {
        NV_V (DV, =, C_STORAGE_R_NV(c,t,SV_PREMIXC_G));
        grad_c = sqrt( NV_DOT(DV, DV) );
    }

    ut = ul*sqrt( 1. + SQR(up/ul) );

    *turb_flame_speed = ut;
    *source = rho_u*ut*grad_c;
}

Hooking a Turbulent Premixed Source UDF to FLUENT

After the UDF that you have defined using DEFINE_TURB_PREMIX_SOURCE is interpreted
or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro
argument will become visible in the User-Defined Function Hooks panel in FLUENT. See
Section 8.2.17 for details on how to hook your DEFINE_TURB_PREMIX_SOURCE UDF to
FLUENT.
4.3 Model-Specific DEFINE Macros

4.3.18 DEFINE_TURBULENT_VISCOSITY

Description

You can use the DEFINE_TURBULENT_VISCOSITY macro to define a custom turbulent viscosity for the Spalart-Allmaras, $k$-$\epsilon$, $k$-$\omega$, and LES turbulence models.

Usage

Macro: DEFINE_TURBULENT_VISCOSITY (name, c, t)

Argument types: cell_t c
Thread *t
Function returns: real

There are three arguments to DEFINE_TURBULENT_VISCOSITY: name, c, and t. You will supply name, the name of the UDF. c and t are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value of the turbulent viscosity to the solver.

The passed variable t is a pointer to the thread on which the turbulent viscosity is to be applied. c is an index that identifies a cell within the given thread.

Example

The following UDF, named user_mu_t, defines a custom turbulent viscosity for the standard $k$-$\epsilon$ turbulence model. Note that the value of M_keCmu in the example is defined through the graphical user interface, but made accessible to all UDFs. The source code can be interpreted or compiled in FLUENT.
UDF that specifies a custom turbulent viscosity for standard k-epsilon formulation using DEFINE_TURBULENT_VISCOSITY

#include "udf.h"

DEFINE_TURBULENT_VISCOSITY(user_mu_t, c, t)
{
    real mu_t;
    real rho = C_R(c,t);
    real k   = C_K(c,t);
    real d   = C_D(c,t);

    mu_t = M_keCu*rho*SQR(k)/d;

    return mu_t;
}

Hooking a Turbulent Viscosity UDF to FLUENT

After the UDF that you have defined using DEFINE_TURBULENT_VISCOSITY is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Viscous Model panel in FLUENT. See Section 8.2.18 for details on how to hook your DEFINE_TURB_VISCOSITY UDF to FLUENT.
4.3.19 DEFINE_UDS_FLUX

Description

You can use the DEFINE_UDS_FLUX macro when you want to customize how the advective flux term is computed in your user-defined scalar (UDS) transport equation. Details on setting up and solving UDS transport equations are provided in Chapter 10.

Usage

Macro: DEFINE_UDS_FLUX (name, f, t, i)

Argument types: face_t f
                 Thread *t
                 int i

Function returns: real

There are four arguments to DEFINE_UDS_FLUX: name, f, t, and i. You will supply name, the name of the UDF. f, t, and i are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value of the mass flow rate through the given face to the solver.

The passed variable t is a pointer to the thread on which the user-defined scalar flux is to be applied. f is an index that identifies a face within the given thread. i is an index that identifies the user-defined scalar.

The advection term in the differential transport equation has the following most general form:

\[ \nabla \cdot \vec{\psi} \phi \]

(4.3-17)

where \( \phi \) is the user-defined scalar conservation quantity and \( \vec{\psi} \) is a vector field. In the default advection term, \( \vec{\psi} \) is, by default, the product of the scalar density and the velocity vector:

\[ \vec{\psi}_{\text{default}} = \rho \vec{v} \]

(4.3-18)

To define the advection term in Equation 4.3-17 using DEFINE_UDS_FLUX, your UDF needs to return the scalar value \( \vec{\psi} \cdot \vec{A} \) to FLUENT, where \( \vec{\psi} \) is the same as defined in Equation 4.3-17 and \( \vec{A} \) is the face normal vector of the face.
Note that the advective flux field that is supplied by your UDF should be divergence-free (i.e., it satisfies the continuity equation). In discrete terms this means that the sum of fluxes over all the faces of each cell should be zero. If the advective field is not divergence-free, then φ is not “conserved” and will result in overshoots/undershoots in the cell value of φ.

You will need to compute \( \vec{\psi} \) in your UDF using, for example, predefined macros for velocity vector and scalar density that Fluent has provided (see Chapter 6 for details) or using your own prescription. The first case is illustrated in the sample C source code, shown below.

Note that if more than one scalar is being solved, you can use a conditional if statement in your UDF to define a different flux function for each \( i \). \( i = 0 \) is associated with scalar-0 (the first scalar equation being solved).

Note also that \( \vec{\psi} \cdot \vec{A} \) must have units of mass flow rate in SI (i.e., kg/s).

```c
real NV_VEC(psi), NV_VEC(A); /* declaring vectors psi and A */
NV_D(psi, =, F_U(f,t), F_V(f,t), F_W(f,t)); /* defining psi in terms of velocity field */
NV_S(psi, *=, F_R(f,t)) /* multiplying density to get psi vector */
F_AREA(A, f, t) /* face normal vector returned from F_AREA */
return NV_DOT(psi, A); /* dot product of the two returned */
```

Additionally, since most quantities in FLUENT are not allocated in memory for interior faces, only for boundary faces (e.g., wall zones), your UDF will also need to calculate interior face values from the cell values of adjacent cells. This is most easily done using the arithmetic mean method. Vector arithmetic can be coded in C using the NV_ and ND_ macros that Fluent has provided (see Chapter 6 for details).

Note that if you had to implement the default advection term in a UDF without the fluid density in the definition of \( \psi \) (see above), you could simply put the following line in your DEFINE_UDS_FLUX UDF:

```
return F_FLUX(f,t) / rho;
```

where the denominator \( \rho \) can be determined by averaging the adjacent cell’s density values \( C_R(F_C0(f,t), \text{THREAD}_0(f,t)) \) and \( C_R(F_C1(f,t), \text{THREAD}_T1(f,t)) \).
Example

The following UDF, named `my_uds_flux`, returns the mass flow rate through a given face. The flux is usually available through the Fluent-supplied macro `F_FLUX(f,t)` (Section 5.4). The sign of flux that is computed by the FLUENT solver is positive if the flow direction is the same as the face area normal direction (as determined by `F_AREA` - see Section 5.2.2), and is negative if the flow direction and the face area normal directions are opposite. By convention, face area normals always point out of the domain for boundary faces, and they point in the direction from cell `c0` to cell `c1` for interior faces.

The UDF must be executed as a compiled UDF.

```c
#******************************************************************************
UDF that implements a simplified advective term in the scalar transport equation
******************************************************************************

#include "udf.h"

DEFINE_UDS_FLUX(my_uds_flux, f, t, i)
{
    Thread *t0, *t1 = NULL;
    cell_t c0, c1 = -1;
    real NV_VEC(psi_vec), NV_VEC(A);

    /* neighboring cells of face f, and their (corresponding) threads */
    t0 = THREAD_T0(f,t);
    c0 = F_C0(f,t);

    if (NULL != THREAD_T1(f,t))
    /* Alternative: if (! BOUNDARY_FACE_THREAD_P(t)) */
    {
        t1 = THREAD_T1(f,t);
        c1 = F_C1(f,t);
    }
    else
    {
        t1 = NULL;
        c1 = -1;
    }

    /* If Face lies at domain boundary, use face values; */
```
DEFINE Macros

/* If Face lies IN the domain, use average of adjacent cells. */

if (NULL == t1)
/* Alternative: if (BOUNDARY_FACE_THREAD_P(t)) */
{
    NV_D(psi_vec, =, F_U(f,t), F_V(f,t), F_W(f,t));
    NV_S(psi_vec, *=, F_R(f,t));
}
else
{
    NV_D(psi_vec, =, C_U(c0,t0), C_V(c0,t0), C_W(c0,t0));
    NV_D(psi_vec, +=, C_U(c1,t1), C_V(c1,t1), C_W(c1,t1));
    NV_S(psi_vec, /=, 2.);  /* averaging. */
    NV_S(psi_vec, *=, (((C_R(c0,t0) + C_R(c1,t1)) / 2.)));)

/* Now psi_vec contains our "psi" from above. */
/* Next, get the face normal vector: */

F_AREA(A, f, t);

/* Finally, return the dot product of both. */
/* Fluent will multiply the returned value */
/* by phi_f (the scalar's value at the face) */
/* to get the "complete" advective term... */

    return NV_DOT(psi_vec, A);
}

Hooking a User-Defined Scalar Flux UDF to FLUENT

After the UDF that you have defined using DEFINE_UDS_FLUX is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Scalars panel in FLUENT. See Section 8.2.19 for details on how to hook your DEFINE_UDS_FLUX UDF to FLUENT.
4.3.20 DEFINE_UDS_UNSTEADY

Description

You can use the DEFINE_UDS_UNSTEADY macro when you want to customize unsteady terms in your user-defined scalar (UDS) transport equations. Details on setting up and solving UDS transport equations are provided in Chapter 10.

Usage

Macro: DEFINE_UDS_UNSTEADY (name, c, t, i, apu, su)

Argument types:

- cell_t c
- Thread *t
- int i
- real *apu
- real *su

Function returns: void

There are six arguments to DEFINE_UDS_UNSTEADY: name, c, t, i, apu, and su. You will supply name, the name of the UDF. c, t, and i are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to set the values of the unsteady terms referenced by the real pointers apu and su to the central coefficient and source term, respectively.

The passed variable t is a pointer to the thread on which the unsteady term for the UDS transport equation is to be applied. c is an index that identifies a cell within the given thread. i is an index that identifies the user-defined scalar for which the unsteady term is to be set.

The FLUENT solver expects that the transient term will be decomposed into a source term, su, and a central coefficient term, apu. These terms are included in the equation set in a similar manner to the way the explicit and implicit components of a source term might be handled. Hence, the unsteady term is moved to the right-hand side and discretized as follows:
unsteady term \[= -\int \frac{\partial}{\partial t} (\rho \phi) \, dV \]
\[\approx -\left[ \frac{(\rho \phi)^n - (\rho \phi)^{n-1}}{\Delta t} \right] \cdot \Delta V \]
\[= -\frac{\rho \Delta V}{\Delta t} \phi^n + \frac{\rho \Delta V}{\Delta t} \phi^{n-1} \] \hspace{1cm} (4.3-19)

Equation 4.3-19 shows how \(su\) and \(apu\) are defined. Note that if more than one scalar is being solved, a conditional if statement can be used in your UDF to define a different unsteady term for each \(i\). \(i = 0\) is associated with scalar-0 (the first scalar equation being solved).

**Example**

The following UDF, named uns_time, modifies user-defined scalar time derivatives using DEFINE_UDS_UNSTEADY. The source code can be interpreted or compiled in FLUENT.

```c
#include "udf.h"

DEFINE_UDS_UNSTEADY(uns_time, c, t, i, apu, su)
{
    real physical_dt, vol, rho, phi_old;
    physical_dt = RP_Get_Real("physical-time-step");
    vol = C_VOLUME(c,t);

    rho = C_R_M1(c,t);
    *apu = -rho*vol / physical_dt; /* implicit part */
    phi_old = C_STORAGE_R(c,t,SV_UDSI_M1(i));
    *su = rho*vol*phi_old/physical_dt; /* explicit part */
}
```

**Hooking an Unsteady UDS Term to FLUENT**

After the UDF that you have defined using DEFINE_UDS_UNSTEADY is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Scalars panel in FLUENT. See Section 8.2.20 for details on how to hook your DEFINE_UDS_UNSTEADY UDF to FLUENT.
4.3.21 DEFINE_VR_RATE

Description

You can use the DEFINE_VR_RATE macro when you want to define a custom volumetric reaction rate for a single reaction or for multiple reactions. During FLUENT execution, DEFINE_VR_RATE is called for every reaction in every single cell.

Usage

Macro:     DEFINE_VR_RATE (name, c, t, r, mw, yi, rr, rr_t)

Argument types:  
cell_t c
Thread *t
Reaction *r
real *mw
real *yi
real *rr
real *rr_t

Function returns:  void

There are eight arguments to DEFINE_VR_RATE: name, c, t, r, mw, yi, rr, and rr_t. You will supply name, the name of the UDF. c, t, r, mw, yi, rr, and rr_t are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to set the values referenced by the real pointers rr and rr_t to the laminar and turbulent reaction rates, respectively.

The passed variable t is a pointer to the thread on which the volumetric reaction rate is to be applied. c is an index that identifies a cell within the given thread. r is a pointer to the data structure that represents the current reaction. mw is a real pointer array of the species molecular weights and yi is a real pointer array of the species mass fractions.

rr and rr_t (defined by the UDF) are computed and the lower of the two values is used when the finite-rate/eddy-dissipation chemical reaction mechanism used. Note that rr and rr_t are conversion rates in kgmol/m^3-s. These rates, when multiplied by the respective stoichiometric coefficients, yield the production/consumption rates of the individual chemical components.
Example 1

The following UDF, named `vol_reac_rate`, specifies a volume reaction rate. The function must be executed as a compiled UDF in FLUENT.

```c
#define VR_RATE(vol_reac_rate, c, t, r, wk, yk, rate, rr_t)
{
    real ci, prod;
    int i;

    /* Calculate Arrhenius reaction rate */
    prod = 1.;
    for(i = 0; i < r->n_reactants; i++)
    {
        ci = C_R(c,t) * yk[r->reactant[i]] / wk[r->reactant[i]];
        prod *= pow(ci, r->exp_reactant[i]);
    }
    *rate = r->A * exp(- r->E / (UNIVERSAL_GAS_CONSTANT * C_T(c,t))) * pow(C_T(c,t), r->b) * prod;
    *rr_rate = *rate;

    /* No "return..;" value. */
}
```
Example 2

When multiple reactions are specified, a volume reaction rate UDF is called several times in each cell. Different values are assigned to the pointer \( r \), depending on which reaction the UDF is being called for. Therefore, you will need to determine which reaction is being called, and return the correct rates for that reaction. Reactions can be identified by their name through the \( r->name \) statement. To test whether a given reaction has the name \texttt{reaction-1}, for example, you can use the following C construct:

\[
\text{if (!strcmp}(r->\text{name}, \text{"reaction-1\})
\{ \\
\text{\hspace{2em} /* r->name is identical to \text{"reaction-1\} */ }
\}
\]

Note that \texttt{strcmp}(r->name, \text{"reaction-1\}) returns 0 which is equal to FALSE when the two strings are identical.

It should be noted that \texttt{DEFINE VR RATE} defines only the reaction rate for a predefined stoichiometric equation (set in the Reactions panel) thus providing an alternative to the Arrhenius rate model. \texttt{DEFINE VR RATE} does not directly address the particular rate of species creation or depletion; this is done by the \texttt{FLUENT} solver using the reaction rate supplied by your UDF.

The following is a source code template that shows how to use \texttt{DEFINE VR RATE} in connection with more than one user-specified reaction. Note that \texttt{FLUENT} always calculates the \texttt{rr} and \texttt{rr_t} reaction rates before the UDF is called. Consequently, the values that are calculated are available only in the given variables when the UDF is called.

```c
/**
 * Multiple reaction UDF that specifies different reaction rates
 * for different volumetric chemical reactions
 */
#include "udf.h"

DEFINE_VR_RATE(myrate, c, t, r, mw, yi, rr, rr_t)
{
    /* If more than one reaction is defined, it is necessary to distinguish
       between these using the names of the reactions. */
    if (!strcmp(r->name, "reaction-1"))
    {
        /* Reaction 1 */
    }
    else if (!strcmp(r->name, "reaction-2"))
```
Hooking a Volumetric Reaction Rate UDF to FLUENT

After the UDF that you have defined using `DEFINE_VR_RATE` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.2.21 for details on how to hook your `DEFINE_VR_RATE` UDF to FLUENT.
4.4 Multiphase DEFINE Macros

The following DEFINE macros are used only for multiphase applications. Table 4.4.1 provides a quick reference guide to the DEFINE macros, the functions they are used to define, and the panels where they are activated in FLUENT. Definitions of each DEFINE macro are listed in the udf.h header file. For your convenience, the definitions are provided in Appendix A.

- DEFINE_CAVITATION_RATE (Section 4.4.1)
- DEFINE EXCHANGE_PROPERTY (Section 4.4.2)
- DEFINE_VECTOR_EXCHANGE_PROPERTY (Section 4.4.3)

Table 4.4.1: Quick Reference Guide for Multiphase DEFINE Macros

<table>
<thead>
<tr>
<th>Model</th>
<th>Function</th>
<th>DEFINE Macro</th>
<th>Panel Activated In</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOF</td>
<td>mass transfer</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
<tr>
<td></td>
<td>mass transfer</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
<tr>
<td></td>
<td>drag coefficient</td>
<td>DEFINE_VECTOR_EXCHANGE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
<tr>
<td></td>
<td>slip velocity</td>
<td>DEFINE_CAVITATION_RATE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td></td>
<td>cavitation rate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mixture</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eulerian</td>
<td>mass transfer</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
<tr>
<td></td>
<td>heat transfer</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
<tr>
<td></td>
<td>drag coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
<tr>
<td></td>
<td>lift coefficient</td>
<td>DEFINE_EXCHANGE_PROPERTY</td>
<td>Phase Interaction</td>
</tr>
</tbody>
</table>
4.4.1 DEFINE_CAVITATION_RATE

Description

You can use the DEFINE_CAVITATION_RATE macro to model the creation of vapor due to pressure tension in a multiphase Mixture model flow.

Usage

Macro: 

```
DEFINE_CAVITATION_RATE (name, c, t, p, rhoV, rhoL, vofV, p_v, n_b, m_dot)
```

Argument types: 

```
cell_t c
Thread *t
real *p
real *rhoV
real *rhoL
real *vofV
real *p_v
real *n_b
real *m_dot
```

Function returns: 

```
void
```

There are ten arguments to DEFINE_CAVITATION_RATE: name, c, t, p, rhoV, rhoL, vofV, p_v, n_b, and m_dot. You will supply name, the name of the UDF. c, t, p, rhoV, rhoL, vofV, p_v, n_b, and m_dot are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to set the value referenced by the real pointer m_dot to the cavitation rate.

The passed variable t is a pointer to the mixture-level thread. c is an index that identifies a cell within the given thread. The remaining arguments are real pointers to the following data: shared pressure (p), vapor density (rhoV), liquid density (rhoL), vapor volume fraction (vofV), vaporization pressure (p_v), number of bubbles per unit volume (n_b), and rate of vapor formation (m_dot)
Example

The following UDF, named user_cavitation_rate, is an example of how to calculate the mass transfer between the liquid and vapor phases (cavitation rate) in a multiphase mixture. The source code can be interpreted or compiled in FLUENT.

```c
#include "udf.h"

#define MIN_VOF 1.e-5
#define MAX_VOF 0.999999

DEFINE_CAVITATION_RATE(user_cavitation_rate, c, t, p, rhoV, rhoL, vofV, p_v, n_b, m_dot)
{
  real p_vapor = *p_v;
  real n_bubbles = *n_b;
  real dp, vofM, radV;
  dp = p_vapor - ABS_P( p[c], op_pres );
  vofM = MIN(MAX(MIN_VOF, vofV[c]),MAX_VOF);
  radV = pow(vofM/((1.-vofM)*4./3.*M_PI*n_bubbles), 1./3.);
  if (dp>0.)
    *m_dot = (1.-vofV[c]) * n_bubbles * 4. * M_PI *
      radV * radV/(1.+n_bubbles*4./3.*M_PI*radV*radV*radV) *
      sqrt(2.*ABS(dp)/(3.*rhoL[c]));
  else
    {
      *m_dot = - (1.-vofV[c]) * n_bubbles * 4. *M_PI *
        radV*radV/(1.+n_bubbles*4./3.*M_PI*radV*radV*radV) *
        sqrt(2.*ABS(dp)/(3.*rhoL[c]));
      if (vofV[c] <= MIN_VOF) *m_dot=0.;
    }
}
```

Hooking a Cavitation Rate UDF to FLUENT

After the UDF that you have defined using DEFINE_CAVITATION_RATE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.3.1 for details on how to hook your DEFINE_CAVITATION_RATE UDF to FLUENT.
### 4.4.2 `DEFINE_EXCHANGE_PROPERTY`

**Description**

You can use the `DEFINE_EXCHANGE_PROPERTY` macro to specify UDFs for phase interaction variables in multiphase models. These include heat and mass transfer functions, as well as drag and lift coefficients. Below is a list of user-defined functions that can be specified using `DEFINE_EXCHANGE_PROPERTY` for each multiphase model in FLUENT. Note that the phase interaction variable surface tension coefficient is defined using `DEFINE_PROPERTY`. See Section 4.3.13 or details.

<table>
<thead>
<tr>
<th>VOF Model</th>
<th>Mixture Model</th>
<th>Eulerian Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>mass transfer</td>
<td>mass transfer</td>
<td>mass transfer</td>
</tr>
<tr>
<td></td>
<td>drag coefficient</td>
<td>heat transfer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drag coefficient</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lift coefficient</td>
</tr>
</tbody>
</table>

**Usage**

**Macro:**

```plaintext```
DEFINE_EXCHANGE_PROPERTY (name, c, mixture_thread, second_column_phase_index, first_column_phase_index)
```

**Argument types:**

- `cell_t c`
- `Thread *mixture_thread`
- `int second_column_phase_index`
- `int first_column_phase_index`

**Function returns:**

`real`

There are five arguments to `DEFINE_EXCHANGE_PROPERTY`: `name`, `c`, `mixture_thread`, `second_column_phase_index`, and `first_column_phase_index`. You will supply `name`, the name of the UDF. `c`, `mixture_thread`, `second_column_phase_index`, and `first_column_phase_index` are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the `real` value of the lift coefficient, drag coefficient, heat or mass transfer to the solver.

`mixture_thread` is a pointer to the mixture-level thread. `c` is the index of a cell on the thread pointed to by `mixture_thread`. `first_column_phase_index` and `second_column_phase_index`...
**4.4 Multiphase DEFINE Macros**

*phase_index* are integer identifiers corresponding to the pair of phases in your multiphase flow that you are specifying a slip velocity for. The identifiers correspond to the phases that are selected in the *Phase Interaction* panel in the graphical user interface. An index of 0 corresponds to the primary phase, and is incremented by one for each secondary phase.

**Example 1 - Drag Coefficient**

The following UDF, named *custom drag*, can be used to customize the default Syamlal drag law in FLUENT. The default drag law uses 0.8 (for void $\leq 0.85$) and 2.65 (void $>0.85$) for $bfac$. This results in a minimum fluid velocity of 25 cm/s. The UDF modifies the drag law to result in a minimum fluid velocity of 8 cm/s, using 0.28 and 9.07 for the $bfac$ parameters.

```c
/***************************************************************
 UDF for customizing the default Syamlal drag law in Fluent
 ***************************************************************

#include "udf.h"

#define pi 4.*atan(1.)
#define diam2 3.e-4

DEFINE_EXCHANGE_PROPERTY(custom_drag, cell, mix_thread, s_col, f_col)
{
    Thread *thread_g, *thread_s;
    real x_vel_g, x_vel_s, y_vel_g, y_vel_s, abs_v, slip_x, slip_y,
        rho_g, rho_s, mu_g, reyp, afac,
        bfac, void_g, vfac, fdrgs, taup, k_g_s;

    /* find the threads for the gas (primary) */
    /* and solids (secondary phases) */

    thread_g = THREAD_SUB_THREAD(mix_thread, s_col); /* gas phase */
    thread_s = THREAD_SUB_THREAD(mix_thread, f_col); /* solid phase */

    /* find phase velocities and properties*/
    x_vel_g = C_U(cell, thread_g);
    y_vel_g = C_V(cell, thread_g);
    x_vel_s = C_U(cell, thread_s);
    y_vel_s = C_V(cell, thread_s);
```

© Fluent Inc. January 22, 2003 4-101
slip_x = x_vel_g - x_vel_s;
slip_y = y_vel_g - y_vel_s;

rho_g = C_R(cell, thread_g);
rho_s = C_R(cell, thread_s);

mu_g = C_MU_L(cell, thread_g);

/*compute slip*/
abs_v = sqrt(slip_x*slip_x + slip_y*slip_y);

/*compute Reynold’s number*/
reyp = rho_g*abs_v*diam2/mu_g;

/* compute particle relaxation time */
taup = rho_s*diam2*diam2/18./mu_g;

void_g = C_VOF(cell, thread_g); /* gas vol frac*/

/*compute drag and return drag coeff, k_g_s*/

afac = pow(void_g,4.14);

if(void_g<=0.85)
  bfac = 0.281632*pow(void_g, 1.28);
else
  bfac = pow(void_g, 9.076960);

vfac = 0.5*(afac-0.06*reyp+sqrt(0.0036*reyp*reyp+0.12*reyp*(2.*bfac-
                                 afac)+afac*afac));
fdrgs = void_g*(pow((0.63*sqrt(reyp)/
                      vfac+4.8*sqrt(vfac/vfac),2))/24.0;

k_g_s = (1.-void_g)*rho_s*fdrgs/taup;

return k_g_s;
}
Example 2 - Heat Transfer Coefficient UDF

The following UDF, named heat_udf, specifies a custom heat transfer coefficient.

```c
/* #include "udf.h" */

#define PR_NUMBER(cp,mu,k) ((cp)*(mu)/(k))
#define IP_HEAT_COEFF(vof,k,nu,d) ((vof)*6.*(k)*(Nu)/(d)/(d))

static real
heat_ranz_marshall(cell_t c, Thread *ti, Thread *tj)
{
    real h;
    real d = C_PHASE_DIAMETER(c,tj);
    real k = C_K_L(c,ti);
    real NV_VEC(v), vel, Re, Pr, Nu;

    NV_DD(v,=,C_U(c,tj),C_V(c,tj),C_W(c,tj),-,C_U(c,ti),C_V(c,ti),C_W(c,ti));
    vel = NV_MAG(v);

    Re = RE_NUMBER(C_R(c,ti),vel,d,C_MU_L(c,ti));
    Pr = PR_NUMBER (C_CP(c,ti),C_MU_L(c,ti),k);
    Nu = 2. + 0.6*sqrt(Re)*pow(Pr,1./3.);

    h = IP_HEAT_COEFF(C_VOF(c,tj),k,Nu,d);
    return h;
}

DEFINE_EXCHANGEPROPERTY(heat_udf, c, t, i, j)
{
    Thread *ti = THREAD_SUB_THREAD(t,i);
    Thread *tj = THREAD_SUB_THREAD(t,j);
    real val;

    val = heat_ranz_marshall(c,ti, tj);
    return val;
}
```
Example 3 - Mass Transfer Coefficient UDF

The following UDF, named \texttt{liq\_gas\_source}, specifies a simple mass transfer coefficient based on saturation temperature.

```c
/* UDF to define a simple mass transfer based on Saturation Temperature */
DEFINE_EXCHANGEPROPERTY(liq_gas_source, cell, thread, liq_i, gas_i)
{
  real m_lg;
  real T_SAT = 373.15.;
  Thread *gas = THREAD_SUB_THREAD(thread, gas_i);
  Thread *liq = THREAD_SUB_THREAD(thread, liq_i);

  m_lg = 0.;
  if (C_T(cell, liq) >= T_SAT)
  {
    m_lg = -0.1*C_VOF(cell,liq)*C_R(cell,liq)*
           fabs(C_T(cell,liq)-T_SAT)/T_SAT;
  }
  if ((m_lg == 0. ) && (C_T(cell, gas) <= T_SAT))
  {
    m_lg = 0.1*C_VOF(cell,gas)*C_R(cell,gas)*
           fabs(T_SAT-C_T(cell,gas))/T_SAT;
  }

  return (m_lg);
}
```

Hooking an Exchange Property UDF to FLUENT

After the UDF that you have defined using \texttt{DEFINE\_EXCHANGE\_PROPERTY} is interpreted or compiled (see Chapter 7 for details), the name that you specified in the \texttt{DEFINE} macro argument will become visible in the Phase Interaction panel in FLUENT. See Section 8.3.2 for details on how to hook your \texttt{DEFINE\_EXCHANGE\_PROPERTY} UDF to FLUENT.
4.4.3 DEFINE VECTOR EXCHANGE PROPERTY

Description

You can use the DEFINE VECTOR EXCHANGE PROPERTY macro to specify custom slip velocities for the multiphase Mixture model.

Usage

Macro:     DEFINE VECTOR EXCHANGE PROPERTY (name, c, mixture_thread, second_column_phase_index, first_column_phase_index, vector_result)

Argument types: cell_t c
                Thread *mixture_thread
                int second_column_phase_index
                int first_column_phase_index
                real *vector_result

Function returns: void

There are six arguments to DEFINE VECTOR EXCHANGE PROPERTY: name, c, mixture_thread, second_column_phase_index, first_column_phase_index, and vector_result. You will supply name, the name of the UDF. c, mixture_thread, second_column_phase_index, first_column_phase_index, and vector_result are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to set the values referenced by the real pointer to the slip velocity vector (vector_result) to the components of the slip velocity vector (e.g., vector_result[0], vector_result[1] for a 2D problem).

The passed variable mixture_thread is a pointer to the mixture-level thread. c is the index of a cell on the thread pointed to by mixture_thread. first_column_phase_index and second_column_phase_index are integer identifiers corresponding to the pair of phases in your multiphase flow that you are specifying a phase interaction slip velocity for. The identifiers correspond to the phases that are defined in the Phase Interaction panel in the graphical user interface. An index of 0 corresponds to the primary phase and is incremented by one for each secondary phase.
Example

The following UDF, named custom_slip, specifies a custom slip velocity in a two-phase mixture problem.

/***************************************************************
UDF for a defining a custom slip velocity in a 2-phase mixture problem
***************************************************************

#include "udf.h"

DEFINE_VECTOR_EXCHANGE_PROPERTY(custom_slip, c, mixture_thread, second_column_phase_index, first_column_phase_index, vector_result)
{
    real K = 5.e4;

    real pgrad_x, pgrad_y;

    Thread *pt, *st;/* thread pointers for primary and secondary phases*/

    pt = THREAD_SUB_THREAD(mixture_thread, second_column_phase_index);
    st = THREAD_SUB_THREAD(mixture_thread, first_column_phase_index);

    /* at this point the phase threads are known for primary (0) and secondary(1) phases */

    pgrad_x = C_DP(c, mixture_thread)[0];
    pgrad_y = C_DP(c, mixture_thread)[1];

    vector_result[0] = -(pgrad_x/K) + (((C_R(c, st)-
        C_R(c, pt))/K)*
        grav[0]);

    vector_result[1] = -(pgrad_y/K) + (((C_R(c, st)-
        C_R(c, pt))/K)*
        grav[1]);
}
4.5 Dynamic Mesh DEFINE Macros

Hooking a Vector Exchange Property UDF to FLUENT

After the UDF that you have defined using DEFINE_VECTOR_EXCHANGE_PROPERTY is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Phase Interaction panel in FLUENT. See Section 8.3.3 for details on how to hook your DEFINE_VECTOR_EXCHANGE_PROPERTY UDF to FLUENT.

4.5 Dynamic Mesh DEFINE Macros

This section contains descriptions of DEFINE macros that you can use to define UDFs that control the behavior of a dynamic mesh. Note that dynamic mesh UDFs that are defined using DEFINE_CG_MOTION, DEFINE_GEOM, and DEFINE_GRID_MOTION can only be executed as compiled UDFs.

Table 4.5.1 provides a quick reference guide to the dynamic mesh DEFINE macros, the functions they define, and the panels where they are activated in FLUENT. Definitions of each DEFINE macro are contained in the udf.h header file. For your convenience, they are listed in Appendix A.

- DEFINE_CG_MOTION (Section 4.5.1)
- DEFINE_GEOM (Section 4.5.2)
- DEFINE_GRID_MOTION (Section 4.5.3)

Table 4.5.1: Quick Reference Guide for Dynamic Mesh-Specific DEFINE Macros

<table>
<thead>
<tr>
<th>Function</th>
<th>DEFINE Macro</th>
<th>Panel Activated In</th>
</tr>
</thead>
<tbody>
<tr>
<td>center of gravity motion</td>
<td>DEFINE_CG_MOTION</td>
<td>Dynamic Zones</td>
</tr>
<tr>
<td>grid motion</td>
<td>DEFINE_GRID_MOTION</td>
<td>Dynamic Zones</td>
</tr>
<tr>
<td>geometry deformation</td>
<td>DEFINE_GEOM</td>
<td>Dynamic Zones</td>
</tr>
</tbody>
</table>
4.5.1 DEFINE(CG,MOTION)

Description

You can use the DEFINE(CG,MOTION) macro to specify the motion of a particular dynamic zone in FLUENT by providing FLUENT with the linear and angular velocities at every time step. FLUENT uses these velocities to update the node positions on the dynamic zone based on solid-body motion. Note that UDFs that are defined using DEFINE(CG,MOTION) can only be executed as compiled UDFs.

Usage

Macro: DEFINE(CG,MOTION) (name, dt, vel, omega, time, dtime)

Argument types: Dynamic_Thread *dt
real vel[]
real omega[]
real time
real dtime

Function returns: void

There are six arguments to DEFINE(CG,MOTION): name, dt, vel, omega, time, and dtime. You will supply name, the name of the UDF. dt, vel, omega, time, and dtime are variables that are passed by the FLUENT solver to your UDF. The linear and angular velocities are returned to FLUENT by overwriting the arrays vel and omega, respectively. dt is a pointer to the structure that stores the dynamic mesh attributes that you have specified (or that are calculated by FLUENT). The current time and time step are provided by FLUENT as time and dtime, respectively.
Example

Consider the following example where the linear velocity is computed from a simple force balance on the body in the x-direction such that

$$\int_{t_0}^{t} dv = \int_{t_0}^{t} (F/m) \ dt$$

(4.5-1)

where $v$ is velocity, $F$ is the force and $m$ is the mass of the body. The velocity at time $t$ is calculated using an explicit Euler formula as

$$v_t = v_{t-\Delta t} + (F/m)\Delta t$$

(4.5-2)

/****************************
* 1-degree of freedom equation of motion (x-direction)
* compiled UDF
*
********************************************/
#include "udf.h"
static real v_prev = 0.0;

DEFINE_CG_MOTION(piston, dt, vel, omega, time, dtime)
{
    Thread *t;
    face_t f;
    real NV_VEC (A);
    real force, dv;

    /* reset velocities */
    NV_S (vel, =, 0.0);
    NV_S (omega, =, 0.0);

    if (!Data_Valid_P ())
        return;

    /* get the thread pointer for which this motion is defined */
    t = DT_THREAD (dt);

    /* compute pressure force on body by looping through all faces */
    force = 0.0;
    begin_f_loop (f, t)
DEFINE Macros

{
    F_AREA (A, f, t);
    force += F_P (f, t) * NV_MAG (A);
}
end_f_loop (f, t)

/* compute change in velocity, i.e., dv = F * dt / mass */
velocity update using explicit Euler formula */
dv = dtime * force / 50.0;
v_prev += dv;
Message ("time = %f, x_vel = %f, force = %f
", time, v_prev, force);

/* set x-component of velocity */
vel[0] = v_prev;
}

Hooking a Center of Gravity Motion UDF to FLUENT

After the UDF that you have defined using DEFINE_CG_MOTION is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Dynamic Zones panel in FLUENT. See Section 8.4.1 for details on how to hook your DEFINE_CG_MOTION UDF to FLUENT.
4.5 Dynamic Mesh DEFINE Macros

4.5.2 DEFINE_GEOM

Description

The DEFINE_GEOM macro is used to define the geometry of a deforming zone. By default, FLUENT provides a mechanism for defining node motion along a planar or cylindrical surface. When FLUENT updates a node on a deforming zone (e.g., through spring-based smoothing or after local face re-meshing) the node is “repositioned” by calling the DEFINE_GEOM UDF. Note that UDFs that are defined using DEFINE_GEOM can only be executed as compiled UDFs.

Usage

Macro: 

```
DEFINE_GEOM (name, d, dt, position)
```

Argument types: 

- char name
- Domain *d
- Dynamic_Thread *dt
- real *position

Function returns: 

void

There are four arguments to DEFINE_GEOM: name, d, dt, and position. You will supply name, the name of the UDF. d, dt, and position are variables that are passed by the FLUENT solver to your UDF. The new position (after projection to the geometry defining the zone) is returned to FLUENT by overwriting the position array.

As in the DEFINE.CG.MOTION macro, dt is a pointer to the dynamic zone data structure, and d is the domain pointer.
Example

The following UDF, named `parabola`, is executed as a compiled UDF.

```c
/****************************/
* defining parabola through points (0, 1), (1/2, 5/4), (1, 1)
* ****************************/
#include "udf.h"

DEFINE_GEOM(parabola, domain, dt, position)
{
  /* set y = -x^2 + x + 1 */
  position[1] = -position[0]*position[0] + position[0] + 1;
}
```

Hooking a Dynamic Mesh Geometry UDF to FLUENT

After the UDF that you have defined using `DEFINE_GEOM` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the Dynamic Zones panel in FLUENT. See Section 8.4.2 for details on how to hook your `DEFINE_GEOM` UDF to FLUENT.
4.5 Dynamic Mesh DEFINE Macros

4.5.3 DEFINE_GRID_MOTION

Description

By default, FLUENT updates the node positions on a dynamic zone by applying the solid-body motion equation. This implies that there is no relative motion between the nodes on the dynamic zone. However, if you need to control the motion of each node independently, then you can use a grid motion UDF. For a DEFINE_GRID_MOTION UDF, you can update the position of each node based on, for example, the deflection due to fluid-structure interaction. Note that UDFs that are defined using DEFINE_GRID_MOTION can only be executed as compiled UDFs.

Usage

Macro:

```
DEFINE_GRID_MOTION (name, d, dt, time, dtime)
```

Argument types:

- char name
- Domain *d
- Dynamic_Thread *dt
- real time
- real dtime

Function returns: void

There are five arguments to DEFINE_GEOM: name, d, dt, time, and dtime. You will supply name, the name of the UDF. d, dt, time, and dtime are variables that are passed by the FLUENT solver to your UDF.

As with the previous UDFs, dt is a pointer to the dynamic mesh structure, d is the domain pointer, time is the current time, and dtime is the time step.
Example

Consider the following example where you want to specify the deflection on a cantilever beam based on the $x$ position such that

$$\omega_y(x) = -10.4\sqrt{x} \sin 26.178 t \quad x > 0.02 \quad (4.5-3)$$

$$\omega_y(x) = 0 \quad x \leq 0.02 \quad (4.5-4)$$

where $\omega_y(x)$ is the $y$-component of the angular velocity at a position $x$. The node position is updated based on

$$(\vec{r})^{t+\Delta t} = (\vec{r})^t + \vec{\Omega} \times (\vec{r})^t \Delta t \quad (4.5-5)$$

where $\vec{\Omega}$ is the angular velocity and $\vec{r}$ is the position vector of a node on the dynamic zone.

```c
#define UDF
#include "udf.h"

DEFINE_GRID_MOTION(beam, domain, dt, time, dtime)
{
    Thread *tf = DT_THREAD (dt);
    face_t f;
    Node *v;
    real NV_VEC (omega), NV_VEC (axis), NV_VEC (dx);
    real NV_VEC (origin), NV_VEC (rvec);
    real sign;
    int n;

    /* set deforming flag on adjacent cell zone */
    SET_DEFORMING_THREAD_FLAG (THREAD_TO (tf));

    sign = -5.0 * sin (26.178 * time);

    Message ("time = %f, omega = %f\n", time, sign);

    /* ... code for updating node position ... */
}
```
4.5 Dynamic Mesh DEFINE Macros

```plaintext
NV_S (omega, =, 0.0);
NV_D (axis, =, 0.0, 1.0, 0.0);
NV_D (origin, =, 0.0, 0.0, 0.152);

begin_f_loop (f, tf)
{
    f_node_loop (f, tf, n)
    {
        v = F_NODE (f, tf, n);

        /* update node if x position is greater than 0.02 and that the current node has not been previously visited when looping through previous faces */
        if (NODE_X (v) > 0.020 && NODE_POS_NEED_UPDATE (v))
        {
            /* indicate that node position has been update so that it’s not updated more than once */
            NODE_POS_UPDATED (v);
            omega[1] = sign * pow (NODE_X (v)/0.230, 0.5);
            NV_VV (rvec, =, NODE_COORD (v), -, origin);
            NV_CROSS (dx, omega, rvec);
            NV_S (dx, *=, dtime);
            NV_V (NODE_COORD (v), +=, dx);
        }
    }
}
end_f_loop (f, tf);
```

**Hooking a DEFINE_GRID_MOTION to FLUENT**

After the UDF that you have defined using DEFINE_GRID_MOTION is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Dynamic Zones panel in FLUENT. See Section 8.4.3 for details on how to hook your DEFINE_GRID_MOTION UDF to FLUENT.
4.6 DPM DEFINE Macros

This section contains descriptions of DEFINE macros for the discrete phase model (DPM). Table 4.6.1 provides a quick reference guide to the DPM DEFINE macros, the functions they define, and the panels where they are activated in FLUENT. Definitions of each DEFINE macro are contained in the udf.h header file. For your convenience, they are listed in Appendix A.

- DEFINE_DPM_BC (Section 4.6.1)
- DEFINE_DPM_BODY_FORCE (Section 4.6.2)
- DEFINE_DPM_DRAG (Section 4.6.3)
- DEFINE_DPM_EROSION (Section 4.6.4)
- DEFINE_DPM_INJECTION_INIT (Section 4.6.5)
- DEFINE_DPM_LAW (Section 4.6.6)
- DEFINE_DPM_OUTPUT (Section 4.6.7)
- DEFINE_DPM_PROPERTY (Section 4.6.8)
- DEFINE_DPM_SCALAR_UPDATE (Section 4.6.9)
- DEFINE_DPM_SOURCE (Section 4.6.10)
- DEFINE_DPM_SPRAY_COLLIDE (Section 4.6.11)
- DEFINE_DPM_SWITCH (Section 4.6.12)
### Table 4.6.1: Quick Reference Guide for DPM-Specific DEFINE Macros

<table>
<thead>
<tr>
<th>Function</th>
<th>DEFINE Macro</th>
<th>Panel Activated In</th>
</tr>
</thead>
<tbody>
<tr>
<td>particle state at boundaries</td>
<td>DEFINE_DPM_BC</td>
<td>boundary condition (e.g., Velocity Inlet)</td>
</tr>
<tr>
<td>body forces on particles</td>
<td>DEFINE_DPM_BODY_FORCE</td>
<td>Discrete Phase Model</td>
</tr>
<tr>
<td>drag coefficients between particles and fluid</td>
<td>DEFINE_DPM_DRAG</td>
<td>Discrete Phase Model</td>
</tr>
<tr>
<td>erosion and accretion rates</td>
<td>DEFINE_DPM_EROSION</td>
<td>Discrete Phase Model</td>
</tr>
<tr>
<td>initializes injections</td>
<td>DEFINE_DPM_INJECTION_INIT</td>
<td>Set Injection Properties</td>
</tr>
<tr>
<td>custom laws for particles</td>
<td>DEFINE_DPM_LAW</td>
<td>Custom Laws</td>
</tr>
<tr>
<td>modifies what is written to the sampling plane output</td>
<td>DEFINE_DPM_OUTPUT</td>
<td>Sample Trajectories</td>
</tr>
<tr>
<td>material properties</td>
<td>DEFINE_DPM_PROPERTY</td>
<td>Materials</td>
</tr>
<tr>
<td>updates scalar every time a particle position is updated</td>
<td>DEFINE_DPM_SCALAR_UPDATE</td>
<td>Discrete Phase Model</td>
</tr>
<tr>
<td>particle source terms</td>
<td>DEFINE_DPM_SOURCE</td>
<td>Discrete Phase Model</td>
</tr>
<tr>
<td>particle collisions algorithm</td>
<td>DEFINE_DPM_SPRAY_COLLIDE</td>
<td>User-Defined Function Hooks</td>
</tr>
<tr>
<td>changes the criteria for switching between laws</td>
<td>DEFINE_DPM_SWITCH</td>
<td>Custom Laws</td>
</tr>
</tbody>
</table>
### 4.6.1 DEFINE_DPM_BC

**Description**

You can use the `DEFINE_DPM_BC` macro if you want to define your own boundary conditions for particles. The function is executed every time a particle touches a boundary of the domain, except for symmetric or periodic boundaries. You can define a separate UDF (using `DEFINE_DPM_BC`) for each boundary.

**Usage**

**Macro:**  
```
DEFINE_DPM_BC (name, p, t, f, f_normal, dim)
```

**Argument types:**  
- `Tracked_Particle *p`
- `Thread *t`
- `face_t f`
- `real f_normal[]`
- `int dim`

**Function returns:**  
`int`

There are six arguments to `DEFINE_DPM_BC`: `name`, `p`, `t`, `f`, `f_normal`, and `dim`. You will supply `name`, the name of the UDF. `p`, `t`, `f`, `f_normal`, and `dim` are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to compute the new velocity of a particle after hitting the wall, and then return the status of the particle track (as an `int`), after it has hit the wall.

The passed variable `p` is a pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked. This pointer can be used as an argument to the particle-specific macros (defined in Section 5.6) to obtain information about particle properties. `t` is a pointer to the face thread the particle is currently hitting. `f` is an index that identifies the face the particle is hitting. `f_normal` is a `real` array that contains the unit vector that is normal to the face. `dim` is the dimension of the flow problem. The value of `dim` is 2 in 2d, for 2d-axisymmetric and 2d-axisymmetric-swirling flow, while it is 3 in 3d flows.

**Example 1**

This example shows the usage of `DEFINE_DPM_BC` for a simple reflection at walls. It is similar to the reflection method executed by FLUENT except that FLUENT accommodates moving walls. The function must be executed as a compiled UDF.

The function assumes an ideal reflection for the normal velocity component (`nor.coef` = 1) while the tangential component is damped (`tan.coef` = 0.3). First, the angle of
incidence is computed. Next, the normal particle velocity, with respect to the wall, is computed and subtracted from the particles velocity. The reflection is complete once the reflected normal velocity is added. The new particle velocity has to be stored in state0 to account for the change of particle velocity in the momentum balance for coupled flows. The function returns PATH_ACTIVE for inert particles while it stops particles of all other types.

/* reflect boundary condition for inert particles */

#include "udf.h"

DEFINE_DPM_BC(bc_reflect, p, t, f, f_normal, dim)
{
    real alpha; /* angle of particle path with face normal */
    real vn=0.;
    real nor_coeff = 1.;
    real tan_coeff = 0.3;
    real normal[3];
    int i, idim = dim;
    real NV_VEC(x);

#if RP_2D
    /* dim is always 2 in 2D compilation. Need special treatment for 2d axisymmetric and swirl flows */
    if (rp_axi_swirl)
    {
        if (R > 1.e-20)
        {
            idim = 3;
            normal[0] = f_normal[0];
            normal[1] = (f_normal[1]*p->state.pos[1])/R;
        }
        else
        {
            for (i=0; i<idim; i++)
                normal[i] = f_normal[i];
        }
    }
    else
#endif

for (i=0; i<idim; i++)
    normal[i] = f_normal[i];

if(p->type==DPM_TYPE_INERT)
{
    alpha = M_PI/2. - acos(MAX(-1.,MIN(1.,NV_DOT(normal,p->state.V)/
                        MAX(NV_MAG(p->state.V),DPM_SMALL))));
    if ((NNULLP(t)) && (THREAD_TYPE(t) == THREAD_F_WALL))
        F_CENTROID(x,f,t);

    /* calculate the normal component, rescale its magnitude by */
    /* the coefficient of restitution and subtract the change */
    /* Compute normal velocity. */
    for(i=0; i<idim; i++)
        vn += p->state.V[i]*normal[i];

    /* Subtract off normal velocity. */
    for(i=0; i<idim; i++)
        p->state.V[i] -= vn*normal[i];

    /* Apply tangential coefficient of restitution. */
    for(i=0; i<idim; i++)
        p->state.V[i] *= tan_coeff;

    /* Add reflected normal velocity. */
    for(i=0; i<idim; i++)
        p->state.V[i] -= nor_coeff*vn*normal[i];

    /* Store new velocity in state0 of particle */
    for(i=0; i<idim; i++)
        p->state0.V[i] = p->state.V[i];

    return PATH_ACTIVE;
}

return PATH_ABORT;
}
Example 2

This example shows how to use DEFINE_DPM_BC for a wall impingement model. The function must be executed as a compiled UDF.

/*
 * Copyright 1988-2002 Fluent Inc and Fluent Deutschland GmbH.
 * All Rights Reserved
 *
 * This is unpublished proprietary source code of Fluent Inc.
 * It is protected by U.S. copyright law as an unpublished work
 * and is furnished pursuant to a written license agreement. It
 * is considered by Fluent Inc. to be confidential and may not be
 * used, copied, or disclosed to others except in accordance with
 * the terms and conditions of the license agreement.
 */

#include "udf.h"
#include "dpm.h"
#include "surf.h"
#include "random.h"

/* define a user-defined dpm boundary condition routine
 * bc_reflect: name
 * p: the tracked particle
 * t: the touched face thread
 * f: the touched face
 * f_normal: normal vector of touched face
 * dim: dimension of the problem (2 in 2d and 2d-axi-swirl, 3 in 3d)
 *
 * return is the status of the particle, see enumeration of Path_Status
 * in dpm.h
 */

#define V_CROSS(a,b,r)
   ((r)[0] = (a)[1]*(b)[2] - (b)[1]*(a)[2],
    (r)[1] = (a)[2]*(b)[0] - (b)[2]*(a)[0],
    (r)[2] = (a)[0]*(b)[1] - (b)[0]*(a)[1])
DEFINE Macros

DEFINE_DPM_BC(bc_wall_jet, p, thread, f, f_normal, dim)
{
    /*
     * Routine implementing the Naber and Reitz Wall
     * impingement model (SAE 880107)
     */

    real normal[3];
    real tan_1[3];
    real tan_2[3];
    real rel_vel[3];
    real face_vel[3];

    real alpha, beta, phi, cp, sp;
    real rel_dot_n, vmag, vnew, dum;
    real weber_in, weber_out;

    int i, idim = dim;

    boolean moving = (SV_ALLOCATED_P(thread,SV_WALL_GRID_V) &&
    SV_ALLOCATED_P(thread,SV_WALL_V ));

#if RP_2D
    if (rp_axi_swirl)
    {
        real R = sqrt(p->state.pos[1]*p->state.pos[1] +
        p->state.pos[2]*p->state.pos[2]);

        if (R > 1.e-20)
        {
            idim = 3;
            normal[0] = f_normal[0];
            normal[1] = (f_normal[1]*p->state.pos[1])/R;
        }
    }
#else
    else
    { for (i=0; i<idim; i++)
        normal[i] = f_normal[i];
    }
#endif
for (i=0; i<idim; i++)
    normal[i] = f_normal[i];

/*
 Set up velocity vectors and calculate the Weber number to determine the regime.
 */

for (i=0; i < idim; i++)
{
    if (moving)
        face_vel[i] = WALL_F_VV(f, thread)[i] + WALL_F_GRID_VV(f, thread)[i];
    else
        face_vel[i] = 0.0;

    rel_vel[i] = P_VEL(p)[i] - face_vel[i];
}

vmag = MAX(NV_MAG(rel_vel), DPM_SMALL);

rel_dot_n = MAX(NV_DOT(rel_vel, normal), DPM_SMALL);

weber_in = P_RHO(p) * DPM_SQR(rel_dot_n) * P_DIAM(p) / MAX( DPM_SURFTEN(p), DPM_SMALL );

/*@ Regime where bouncing occurs (We_in < 80).
(Data from Mundo, Sommerfeld and Tropea
 Int. J. of Multiphase Flow, v21, #2, pp151-173, 1995)
*/

if (weber_in <= 80.)
{
    weber_out = 0.6785*weber_in*exp(-0.04415*weber_in);
    vnew = rel_dot_n * (1.0 + sqrt( weber_out / MAX( weber_in, DPM_SMALL )));

    /* The normal component of the velocity is changed based on the experimental paper above (i.e. the Weber number is based on the relative velocity).
*/
for (i=0; i < idim; i++)
P_VEL(p)[i] = rel_vel[i] - vnew*normal[i] + face_vel[i];

if (weber_in > 80.)
{
alpha = acos(-rel_dot_n/vmag);

/*
Get one tangent vector by subtracting off the normal component from the impingement vector, then cross the normal with the tangent to get an out of plane vector.
*/

for (i=0; i < idim; i++)
tan_1[i] = rel_vel[i] - rel_dot_n*normal[i];

UNIT_VECT(tan_1,tan_1);

V_CROSS(tan_1,normal,tan_2);

/*
beta is calculated by neglecting the coth(alpha) term in the paper (it is approximately right).
*/

beta = MAX(M_PI*sqrt(sin(alpha)/(1.0-sin(alpha))),DPM_SMALL);

phi = -M_PI/beta*log(1.0-cheap_uniform_random()*(1.0-exp(-beta)));

if (cheap_uniform_random() > 0.5)
phi = -phi;

vnew = vmag;

cp = cos(phi);
sp = sin(phi);

for (i=0; i < idim; i++)
P_VEL(p)[i] = vnew*(tan_1[i]*cp + tan_2[i]*sp) + face_vel[i];
}
/ * Subtract off from the original state.  */
  for ( i=0; i < idim; i++ )
    P_VEL0(p)[i] = P_VEL(p)[i];

if ( DPM_STOCHASTIC_P(p->injection) )
{
  /* Reflect turbulent fluctuations also */
  /* Compute normal velocity. */
  dum = 0;
  for( i=0; i<idim; i++ )
    dum += p->V_prime[i]*normal[i];
  /* Subtract off normal velocity. */
  for( i=0; i<idim; i++ )
    p->V_prime[i] -= 2.*dum*normal[i];
  return PATH_ACTIVE;
}

Hooking a DPM Boundary Condition UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_BC is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the appropriate boundary condition panel (e.g., the Velocity Inlet panel) in FLUENT. See Section 8.5.1 for details on how to hook your DEFINE_DPM_BC UDF to FLUENT.
4.6.2 DEFINE_DPM_BODY_FORCE

Description

You can use the DEFINE_DPM_BODY_FORCE macro to specify a body force other than a gravitational or drag force on the particles.

Usage

Macro: DEFINE_DPM_BODY_FORCE (name, p, i)

Argument types: Tracked_Particle *p
int i

Function returns: real

There are three arguments to DEFINE_DPM_BODY_FORCE: name, p, and i. You will supply name, the name of the UDF. p and i are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value of the acceleration due to the body force (in m/s^2) to the FLUENT solver.

The passed variable p is a pointer to a Tracked_Particle structure which contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties). i is an index (0, 1, or 2) that identifies the Cartesian component of the body force that is to be returned by the function.

Example

The following UDF, named particle_body_force, computes the magnetic force on a charged particle. DEFINE_DPM_BODY_FORCE is called at every particle time step in FLUENT and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

In the UDF presented below a charged particle is introduced upstream, into a laminar flow, and travels downstream until \( t = t_{\text{start}} \) when a magnetic field is applied. The particle takes on an approximately circular path (not an exact circular path, because the speed and magnetic force vary as the particle is slowed by the surrounding fluid).

The macro P_TIME(p) gives the current time for a particle traveling along a trajectory, which is pointed to by p.
/* UDF for computing the magnetic force on a charged particle */

#include "udf.h"

#define Q 1.0    /* particle electric charge     */
#define BZ 3.0   /* z component of magnetic field */
#define TSTART 18.0 /* field applied at t = tstart */

/* Calculate magnetic force on charged particle. Magnetic */
/* force is particle charge times cross product of particle*/
/* velocity with magnetic field: Fx= q*bz*Vy, Fy= -q*bz*Vx */

DEFINE_DPM_BODY_FORCE(particle_body_force, p, i)
{
    real bforce;
    if(P_TIME(p) >= TSTART)
        {
            if(i==0) bforce=Q*BZ*P_VEL(p)[1];
            else if(i==1) bforce=-Q*BZ*P_VEL(p)[0];
        }
    else
        bforce=0.0;
    /* an acceleration should be returned */
    return (bforce/P_MASS(p));
}

Hooking a DPM Body Force UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_BODY_FORCE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Discrete Phase Model panel in FLUENT. See Section 8.5.2 for details on how to hooked your DEFINE_DPM_BODY_FORCE UDF to FLUENT.
4.6.3 DEFINE_DPM_DRAG

Description

You can use the DEFINE_DPM_DRAG macro to specify the drag coefficient, $C_D$, between particles and fluid defined by the following equation:

$$F_D = \frac{18\mu C_D Re}{\rho_p D_p^2} \frac{24}{24}$$

Usage

Macro: DEFINE_DPM_DRAG (name, Re, p)

Argument types: real Re
                 Tracked_Particle *p

Function returns: real

There are three arguments to DEFINE_DPM_DRAG: name, Re, and p. You will supply name, the name of the UDF. Re and p are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to return the real value of the drag force on a particle. The value returned to the solver must be dimensionless and represent $18 \times C_d \times Re / 24$.

The passed variable Re is the particle Reynolds number based on the particle diameter and relative gas velocity. p is a pointer to a Tracked_Particle structure which contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties).

Example

The following UDF, named particle_drag_force, computes the drag force on a particle and is a variation of the body force UDF presented in Section 4.6.2. The flow is the same, but a different curve is used to describe the particle drag. DEFINE_DPM_DRAG is called at every particle time step in FLUENT, and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.
UDF for computing particle drag coefficient (18 Cd Re/24) curve as suggested by R. Clift, J. R. Grace and M.E. Weber
"Bubbles, Drops, and Particles" (1978)
************************************************************************/
#include "udf.h"

DEFINE_DPM_DRAG(particle_drag_force, Re, p)
{
  real w, drag_force;

  if (Re < 0.01)
    {
      drag_force=18.0;
      return (drag_force);
    }
  else if (Re < 20.0)
    {
      w = log10(Re);
      drag_force = 18.0 + 2.367*pow(Re,0.82-0.05*w) ;
      return (drag_force);
    }
  else /* Note: suggested valid range 20 < Re < 260 */
    {
      drag_force = 18.0 + 3.483*pow(Re,0.6305) ;
      return (drag_force);
    }
}

Hooking a DPM Drag Coefficient UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_DRAG is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Discrete Phase Model panel in FLUENT. See Section 8.5.3 for details on how to hook your DEFINE_DPM_DRAG UDF to FLUENT.
4.6.4 DEFINE_DPM_EROSION

Description

You can use the DEFINE_DPM_EROSION macro to access the erosion and accretion rates calculated as the particle stream strikes a wall surface. The function is called when the particle encounters a reflecting surface.

Usage

Macro: DEFINE_DPM_EROSION (name, p, t, f, normal, alpha, Vmag, mdot)

Argument types: Tracker Particle *p
Thread *t
face_t f
real normal[]
real alpha
real Vmag
real mdot

Function returns: void

There are eight arguments to DEFINE_DPM_EROSION: name, p, t, f, normal, alpha, Vmag, and mdot. You will supply name, the name of the UDF. p, t, f, normal, alpha, Vmag, and mdot are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to compute the values for the erosion rate and/or accretion rate and store the values at the faces in F_STORAGE_R(f, t, SV_DPM_EROSION) and F_STORAGE_R(f, t, SV_DPM_ACRETION).

The passed variable p is a pointer to a Tracker Particle structure which contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties).

t is a pointer to the thread that the particle is hitting. f is an index that identifies the face the particle is hitting. normal is a real array that contains the unit vector that is normal to the face. alpha is a real variable that represents the impact angle between the particle path and the face (in radians). Vmag is a real variable that represents the magnitude of the particle velocity (in m/s). mdot is the real flow rate of the particle stream as it hits the face (in kg/s).
Example

The following is an example of a compiled UDF that uses DEFINE_DPM_EROSION to extend post-processing of wall impacts in a 2D axisymmetric flow. It provides additional information on how the local particle deposition rate depends on the diameter and normal velocity of the particles. It is based on the assumption that every wall impact leads to more accretion, and, therefore, every trajectory is “evaporated” at its first wall impact. (This is done by first setting a DPM user scalar within DEFINE_DPM_EROSION, which is then evaluated within DEFINE_DPM_LAW, where P_MASS is set to zero.) User-defined memory locations (UDMLs) are used to store and visualize the following:

- number of wall impacts since UDMLs were reset. (Resetting is typically done at the beginning of a FLUENT session by the use of DEFINE_ON_DEMAND in order to avoid the use of uninitialized data fields. Resetting prevents the addition of sampled data being read from a file.)
- average diameter of particles hitting the wall.
- average radial velocity of particles.

Before tracing the particles, you will have to reset the UDMLs and assign the global domain pointer by executing the DEFINE_ON_DEMAND function.

/**************************************************************************
 UDF for extending post-processing of wall impacts
***************************************************************************/
#include "udf.h"

#define MIN_IMPACT_VELO -1000.

/* Minimum particle velocity normal to wall (m/s) to allow Accretion. */

Domain *domain; /* Get the domain pointer and assign it later to domain*/

enum /* Enumeration of used User-Defined Memory Locations. */
{
    NUM_OF_HITS, /* Number of particle hits into wall face considered. */
    AVG_DIAMETER, /* Average diameter of particles that hit the wall. */
    AVG_RADI_VELO, /* Average radial velocity of "" "" " " " " " " " " " " " " " " " */
    NUM_OF_USED_UDM
};

int UDM_checked = 0; /* Availability of UDMLs checked? */
void reset_UDM_s(void); /* Function to follow below. */

int
check_for_UDM(void) /* Check for UDMLs’ availability... */
{
    Thread *t;

    if (UDM_checked)
        return UDM_checked;

    if (!rp_axi)
        Internal_Error("UDF-Error: only valid for 2d-axisymmetric cases!\n");

    thread_loop_c(t, domain) /* We require all cell threads to.. */
    {
        /* provide space in memory for UDML */
        if (FLUID_THREAD_P(t))
            if (NULLP(THREAD_STORAGE(t,SV_UDM_I)))
                return 0;
    }

    UDM_checked = 1; /* To make the following work properly... */
    reset_UDM_s(); /* This line will only be executed once, */
    return UDM_checked; /* because check_for_UDM checks... */
} /* ...for UDM_checked first. */

void
reset_UDM_s(void)
{
    Thread *t;
    cell_t  c;
    face_t  f;
    int     i;

    if (!check_for_UDM()) /* Don’t do it, if memory is not available. */
        return;

    Message("Resetting User Defined Memory...\n");
thread_loop_f(t, domain)
{
    if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
    {
        begin_f_loop(f, t)
        {
            for (i = 0; i < NUM_OF_USED_UDM; i++)
                F_UDMI(f, t, i) = 0.;
        }
        end_f_loop(f, t)
    }
    else
    {
        Message(" Skipping FACE thread no. %d..\n", THREAD_ID(t));
    }
}

thread_loop_c(t, domain)
{
    if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
    {
        begin_c_loop(c, t)
        {
            for (i = 0; i < NUM_OF_USED_UDM; i++)
                C_UDMI(c, t, i) = 0.;
        }
        end_c_loop(c, t)
    }
    else
    {
        Message(" Skipping CELL thread no. %d..\n", THREAD_ID(t));
    }
} /* Skipping Cell Threads can happen if the user */
    /* uses reset_UDM prior to initializing. */
    Message(" --- Done..\n");
}

DEFINE_DPM_SCALAR_UPDATE(dpm_scalup, c, t, if_init, p)
{ if (if_init)
    P_USER_REAL(p, 0) = 0; /* Simple initialization. Used later for */
        /* stopping trajectory calculation */
}
DEFINE Macros

DEFINE_DPM_EROSION(dpm_accr, p, t, f, normal, alpha, Vmag, Mdot)
{
    real A[ND_ND], area;
    int num_in_data;
    Thread *t0;
    cell_t c0;

    real radi_pos[2], radius, imp_vel[2], vel_ortho;

    /* The following is ONLY valid for 2d-axisymmetric calculations!!! */
    /* Additional effort is necessary because DPM tracking is done in */
    /* THREE dimensions for TWO-dimensional axisymmetric calculations. */
    radi_pos[0] = p->state.pos[1]; /* Radial location vector. */
    radi_pos[1] = p->state.pos[2]; /* (Y and Z in 0 and 1...) */

    radius = NV_MAG(radi_pos);
    NV_VS(radi_pos, =, radi_pos, /, radius);
        /* Normalized radius direction vector.*/
    imp_vel[0] = P_VEL(p)[0]; /* Axial particle velocity component. */
    imp_vel[1] = NVD_DOT(radi_pos, P_VEL(p)[1], P_VEL(p)[2], 0.);
        /* Dot product of normalized radius vector and y & z components */
        /* of particle velocity vector gives _radial_ particle velocity */
        /* component */
    vel_ortho = NV_DOT(imp_vel, normal); /*velocity orthogonal to wall */

    if (vel_ortho < MIN_IMPACT_VELO) /* See above, MIN_IMPACT_VELO */
        return;

    if (!UDM_checked) /* We will need some UDM’s, */
        if (!check_for_UDM()) /* so check for their availability.. */
            return; /* (Using int variable for speed, could */
            /* even just call check_for UDFM().) */
    c0 = F_CO0(f, t);
    t0 = THREAD_T0(f, t);

    num_in_data = F_UDMI(f, t, NUM_OF_HITS);

    /* Add particle to statistics: Calculate...:
    current_particle_property +
    earlier_particles_averaged_property * number_of_earlier_particles
    ----------------------------------------------------------
    number_of_earlier_particles + 1 */
4.6 DPM DEFINE Macros

/* Average diameter of particles that hit the particular wall face: */
F_UDMI(f, t, AVG_DIAMETER) = (P_DIAM(p) + num_in_data * F_UDMI(f, t, AVG_DIAMETER)) / (num_in_data + 1);
C_UDMI(c0, t0, AVG_DIAMETER) = F_UDMI(f, t, AVG_DIAMETER);

/* Average velocity normal to wall of particles hitting the wall: */
F_UDMI(f, t, AVG_RADI_VELO) = (vel_ortho + num_in_data * F_UDMI(f, t, AVG_RADI_VELO)) / (num_in_data + 1);
C_UDMI(c0, t0, AVG_RADI_VELO) = F_UDMI(f, t, AVG_RADI_VELO);

F_UDMI(f, t, NUM_OF_HITS) = num_in_data + 1;
C_UDMI(c0, t0, NUM_OF_HITS) = num_in_data + 1;

F_AREA(A, f, t);
area = NV_MAG(A);
F_STORAGE_R(f, t, SV_DPMS_ACCRETION) += Mdot / area;
/* copied from source. */

P_USER_REAL(p, 0) = 1.; /* "Evaporate" */

DEFINE_DPM_LAW(stop_dpm_law, p, if_cpld)
{
    if (0. < P_USER_REAL(p, 0))
        P_MASS(p) = 0.;  /* "Evaporate" */
}

DEFINE_ON_DEMAND(reset_UDM)
{
    /* assign domain pointer with global domain */
    domain = Get_Domain(1);
    reset_UDM_s();
}

Hooking an Erosion/Accretion UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_EROSION is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Discrete Phase Model panel in FLUENT. See Section 8.5.4 for details on how to hook your DEFINE_DPM_EROSION UDF to FLUENT.
4.6.5 DEFINE_DPM_INJECTION_INIT

Description

You can use the DEFINE_DPM_INJECTION_INIT macro when you want to initialize a particle's injection properties such as location, diameter, and velocity.

Usage

Macro:  

```
DEFINE_DPM_INJECTION_INIT (name, I)
```

Argument types: Injection *I

Function returns: void

There are two arguments to DEFINE_DPM_INJECTION_INIT: name and I. You will supply name, the name of the UDF. I is a variable that is passed by the FLUENT solver to your UDF.

The passed variable I is a pointer to the Injection structure which is a container for the particles being created. This function is called twice for each Injection before the first DPM iteration, and then called once for each Injection before the particles are injected into the domain at each subsequent DPM iteration.

Example

The following UDF, named init_bubbles, initializes particles on a surface injection due to a surface reaction. This function must be executed as a compiled UDF. Note that if you are going to use this UDF in a transient simulation to compute transient particles, you will need to replace `loop(p, I->p)` with `loop(p, I->p_init)`. Transient particle initialization cannot be performed with a loop over `I->p`. 
/************/ UDF that initializes particles on a surface injection due to a surface reaction /************/ 

#include "udf.h"

#define REACTING_SURFACE_ID 2
#define MW_H2 2
#define STOIC_H2 1
#define P_CELL(P) RP_CELL(&((P)->cCell)) /* Non-standard macros */
#define P_CELL_THREAD(P) RP_THREAD(&((P)->cCell))

real contact_area(cell_t c, Thread *t, int s_id, int *n);

DEFINE_DPM_INJECTION_INIT(init_bubbles,I) {
  int count,i;
  real area, mw[MAX_SPE_EQNS], yi[MAX_SPE_EQNS];
  /* MAX_SPE_EQNS is a Fluent constant in materials.h */

  Particle *p;
  cell_t cell;
  Thread *cthread;
  Material *mix, *sp;

  Message("Initializing Injection: %s\n",I->name);

  loop(p,I->p) /* Standard Fluent Looping Macro to get particle streams in an Injection */ {
    cell = P_CELL(p); /* Get the cell and thread that the particle is currently in */
    cthread = P_CELL_THREAD(p);

    /* Set up molecular weight & mass fraction arrays */
    mix = THREAD_MATERIAL(cthread);
    mixture_species_loop(mix,sp,i) {
      mw[i] = MATERIAL_PROP(sp,PROP_mwi);
      yi[i] = C_YI(cell,cthread,i);
    }
area = contact_area(cell, cthread, REACTING_SURFACE_ID,&count);
/* Function that gets total area of REACTING_SURFACE faces in contact with cell */
/* count is the number of contacting faces, and is needed to share the total bubble emission between the faces */
if (count > 0) /* if cell is in contact with REACTING_SURFACE */
{
    P_FLOW_RATE(p) = (area *MW_H2* STOIC_H2 * reaction_rate(cell, cthread, mw, yi))/(real)count; /* to get correct total flow rate when multiple faces contact the same cell */
    P_DIAM(p) = 1e-3;
    P_RHO(p) = 1.0;
    P_MASS(p) = P_RHO(p)*M_PI*pow(P_DIAM(p),3.0)/6.0;
}
else
    P_FLOW_RATE(p) = 0.0;
}

real contact_area(cell_t c, Thread *t, int s_id, int *n)
{
    int i = 0;
    real area = 0.0, A[ND_ND];

    *n = 0;
    c_face_loop(c,t,i)
    {
        if(THREAD_ID(C_FACE_THREAD(c,t,i)) == s_id)
        {
            (*n)++;
            F_AREA(A,C_FACE(c,t,i), C_FACE_THREAD(c,t,i));
            area += NV_MAG(A);
        }
    }
}

Hooking a DPM Initialization UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_INJECTION_INIT is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Set Injection Properties panel in FLUENT. See Section 8.5.5 for details on how to hook your DEFINE_DPM_INJECTION_INIT UDF to FLUENT.
4.6.6 DEFINE_DPM_LAW

Description

You can use the DEFINE_DPM_LAW macro when you want to customize laws for particles. For example, your UDF can specify custom laws for heat and mass transfer rates for droplets and combusting particles. Additionally, you can specify custom laws for mass, diameter, and temperature properties as the droplet or particle exchanges mass and energy with its surroundings.

Usage

Macro: DEFINE_DPM_LAW (name, p, ci)

Argument types: Tracked_Particle *p
                int ci

Function returns: void

There are three arguments to DEFINE_DPM_LAW: name, p, and ci. You will supply name, the name of the UDF. p and ci are variables that are passed by the FLUENT solver to your UDF.

The passed variable p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties). ci is an integer that indicates whether the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).
Example

The following UDF, named Evapor_Swelling_Law, models a custom law for the evaporation swelling of particles. The source code can be interpreted or compiled in FLUENT. See Section 4.6.12 for another example of DEFINE_DPM_LAW usage.

```c
#include "udf.h"

DEFINE_DPM_LAW(Evapor_Swelling_Law, p, ci)
{
    real swelling_coeff = 1.1;

    /* first, call standard evaporation routine to calculate *
       * the mass and heat transfer
       */
    VaporizationLaw(p);
    /* compute new particle diameter and density */
    P_DIAM(p) = P_INIT_DIAM(p)*(1. + (swelling_coeff - 1.)*
        (P_INIT_MASS(p)-P_MASS(p))/(DPM_VOLATILE_FRACTION(p)*P_INIT_MASS(p)));
    P_RHO(p) = P_MASS(p) / (3.14159*P_DIAM(p)*P_DIAM(p)*P_DIAM(p)/6);
    P_RHO(p) = MAX(0.1, MIN(1e5, P_RHO(p)));
}
```

Hooking a Custom DPM Law to FLUENT

After the UDF that you have defined using DEFINE_DPM_LAW is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Custom Laws panel in FLUENT. See Section 8.5.6 for details on how to hook your DEFINE_DPM_LAW UDF to FLUENT.
4.6 DPM DEFINE Macros

4.6.7 DEFINE_DPM_OUTPUT

Description

You can use the DEFINE_DPM_OUTPUT macro when you want to modify what is written to the sampling device output. This function allows access to the variables that are written as a particle passes through a sampler (see Chapter 20 of the User’s Guide).

Usage

Macro: DEFINE_DPM_OUTPUT (name, header, fp, p, t, plane)

Argument types: int header
FILE *fp
Tracked_Particle *p
Thread *t
Plane *plane

Function returns: void

There are six arguments to DEFINE_DPM_OUTPUT: name, header, fp, p, t, and plane. You will supply name, the name of the UDF. header, fp, p, t, and plane are variables that are passed by the FLUENT solver to your UDF. The output of your UDF will be written to the file indicated by fp.

The passed variable header is an integer that is equal to 1 at the first call of the function before particles are tracked and set to 0 for subsequent calls. fp is a pointer to the file to or from which you are writing or reading. p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties).

t is a pointer to the thread that the particle is passing through if the sampler is represented by a grid surface. If the sampler is not defined as a grid surface, then the value of t is NULL. plane is a pointer to the Plane structure (see dpm.h) if the sampling device is defined as a planar slice (line in 2d). If a grid surface is used by the sampler, then plane is NULL.
Example

See Section 4.6.9 for an example of a UDF that uses the `DEFINE_DPM_OUTPUT` macro.

Hooking a DPM Output UDF to FLUENT

After the UDF that you have defined using `DEFINE_DPM_OUTPUT` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the Sample Trajectories panel in FLUENT. See Section 8.5.7 for details on how to hook your `DEFINE_DPM_OUTPUT` UDF to FLUENT.
4.6.8 DEFINE_DPM_PROPERTY

Description

You can use the DEFINE_DPM_PROPERTY macro when you want to specify properties of discrete phase materials. For example, you can model the following dispersed phase properties with this type of UDF:

- particle emissivity
- vapor pressure
- vaporization temperature
- particle scattering factor
- boiling point
- particle viscosity
- particle surface tension

Usage

Macro: DEFINE_DPM_PROPERTY (name, c, t, p)

Argument types: cell_t c
Thread *t
Tracked_Particle *p

Function returns: real

There are four arguments to DEFINE_DPM_PROPERTY: name, c, t, and p. DEFINE_DPM_PROPERTY has the same arguments as the DEFINE_PROPERTY function (described in Section 4.3.13), with the addition of the pointer to the Tracked_Particle p. You will supply name, the name of the UDF. c, t, and p are variables that are passed by the FLUENT solver to your UDF. Your UDF will need to compute the real value of the discrete phase property and return it to the solver.

The passed variable c is an index that identifies the cell where the particle is located in the given thread. t is a pointer to the thread where the particle is located. p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties).
Example

In the following example, two discrete phase material property UDFs (named coal_emissivity and coal_scattering, respectively) are concatenated into a single C source file. These UDFs must be executed as compiled UDFs in FLUENT.

/*******************************************************************************/
UDF that specifies discrete phase materials
*******************************************************************************/

#include "udf.h"

DEFINE_DPM_PROPERTY(coal_emissivity, c, t, p)
{
    real mp0= P_INIT_MASS(p);
    real mp = P_MASS(p);
    real vf, cf;

    /* get the material char and volatile fractions and store them */
    /* in vf and cf */
    vf=DPM_VOLATILE_FRACTION(p);
    cf=DPM_CHAR_FRACTION(p);

    if (!(((mp/mp0) >= 1) || ((mp/mp0) <= 0)))
    {
        if ((mp/mp0) < (1-(vf)-(cf)))
        {
            /* only ash left */
            /* vf = cf = 0; */
            return .001;
        }
        else if ((mp/mp0) < (1-(vf)))
        {
            /* only ash and char left */
            /* cf = 1 - (1-(vf)-(cf))/(mp/mp0); */
            /* vf = 0; */
            return 1.0;
        }
    }
else
{
    /* volatiles, char, and ash left */
    /* cf = (cf)/(mp/mp0); */
    /* vf = 1. - (1.-(vf))/(mp/mp0); */
    return 1.0;
}

DEFINE_DPM_PROPERTY(coal_scattering, c, t, p)
{
    real mp0= P_INIT_MASS(p);
    real mp = P_MASS(p);
    real cf, vf;

    /* get the original char and volatile fractions and store them */
    /* in vf and cf */
    vf=DPM_VOLATILE_FRACTION(p);
    cf=DPM_CHAR_FRACTION(p);

    if (!(mp/mp0 >= 1) || ((mp/mp0) <= 0))
    {
        if ((mp/mp0) < (1-(vf)-(cf)))
        {
            /* only ash left */
            /* vf = cf = 0; */
            return 1.1;
        }
        else if ((mp/mp0) < (1-(vf)))
        {
            /* only ash and char left */
            /* cf = 1 - (1-(vf)-(cf))/(mp/mp0); */
            /* vf = 0; */
            return 0.9;
        }
    }
else
{
    /* volatiles, char, and ash left */
    /* cf = (cf)/(mp/mp0); */
    /* vf = 1. - (1.-(vf))/(mp/mp0); */
    return 1.0;
}

return 1.0;

---

Hooking a DPM Material Property UDF to FLUENT

After the UDF that you have defined using `DEFINE_DPM_PROPERTY` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the Materials panel in FLUENT. See Section 8.5.8 for details on how to hook your `DEFINE_DPM_PROPERTY` UDF to FLUENT.
4.6.9  DEFINE_DPM_SCALAR_UPDATE

Description

You can use the DEFINE_DPM_SCALAR_UPDATE macro to update scalar quantities every time a particle position is updated. The function allows particle-related variables to be updated or integrated over the life of the particle. Particle values can be stored in an array associated with the Tracked_Particle (accessed with the macro P_USER_REAL(p,i)). Values calculated and stored in the array can be used to color the particle trajectory.

During FLUENT execution, the DEFINE_DPM_SCALAR_UPDATE function is called at the start of particle integration (when initialize is equal to 1) and then after each time step for the particle trajectory integration.

Usage

Macro:         DEFINE_DPM_SCALAR_UPDATE (name, c, t, initialize, p)

Argument types:  cell_t c
                 Thread *t
                 int initialize
                 Tracked_Particle *p

Function returns:  void

There are five arguments to DEFINE_DPM_SCALAR_UPDATE: name, c, t, initialize, and p. You will supply name, the name of the UDF. c, t, initialize, and p are variables that are passed by the FLUENT solver to your UDF.

The passed variable c is an index that identifies the cell that the particle is currently in. t is a pointer to the thread the particle is currently in. initialize is an integer that has a value of 1 when the function is called at the start of the particle integration, and 0 thereafter. p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties). The real array user is available for storage. The size of this array should be set in the Discrete Phase Model panel in the Number of Scalars field.
Example

The following compiled UDF computes the melting index along a particle trajectory. The DEFINE_DPM_SCALAR_UPDATE function is called at every particle time step in FLUENT and requires a significant amount of CPU time to execute.

The melting index is computed from

\[
\text{melting index} = \int_0^t \frac{1}{\mu} dt
\]  

(4.6-1)

Also included in this UDF is an initialization function DEFINE_INIT that is used to initialize the scalar variables. DPM_OUTPUT is used to write the melting index at sample planes and surfaces. The macro NULL_P, which expands to \((p) == \text{NULL}\), checks if its argument is a null pointer.

/*****************************************************************************
 * UDF for computing the melting index along a particle trajectory
*******************************************************************************/
#include "udf.h"

static real viscosity_0;

DEFINE_INIT(melt_setup, domain)
{
    /* if memory for the particle variable titles has not been
     * allocated yet, do it now */

    if (NULLP(user_particle_vars)) Init_User_Particle_Vars();

    /* now set the name and label */

    strcpy(user_particle_vars[0].name,"melting-index");
    strcpy(user_particle_vars[0].label,"Melting Index");
}

    /* update the user scalar variables */
DEFINE_DPM_SCALAR_UPDATE(melting_index, cell, thread, initialize, p)
{
    cphase_state_t *c = &(p->cphase);
    if (initialize)
    {
        /* this is the initialization call, set:
        * p->user[0] contains the melting index, initialize to 0
        * viscosity_0 contains the viscosity at the start of a time step*/

        p->user[0] = 0.;
        viscosity_0 = c->mu;
    }
    else
    {
        /* use a trapezoidal rule to integrate the melting index */
        p->user[0] += P_DT(p) * .5 * (1/viscosity_0 + 1/c->mu);

        /* save current fluid viscosity for start of next step */
        viscosity_0 = c->mu;
    }
}

/* write melting index when sorting particles at surfaces */
DEFINE_DPM_OUTPUT(melting_output, header, fp, p, thread, plane)
{
    char name[100];

    if (header)
    {
        if (NNULLP(thread))
            printf(fp,"(%s %d)\n",thread->head->dpm_summary.sort_file_name,11);
        else
            printf(fp,"(%s %d)\n",plane->sort_file_name,11);

        printf(fp,"(%10s %10s %10s %10s %10s %10s %10s %10s %10s %10s %s)\n",
                "X","Y","Z","U","V","W","diameter","T","mass-flow",
                "time","melt-index","name");
    }
}
else
{
    sprintf(name,"%s:%d",p->injection->name,p->part_id);
    printf(fp,
            "((%10.6g %10.6g %10.6g %10.6g %10.6g %10.6g \\
             %10.6g %10.6g %10.6g %10.6g %10.6g) %s)\n",
             p->state.pos[0], p->state.pos[1], p->state.pos[2],
             p->state.V[0], p->state.V[1], p->state.V[2],
             p->state.diam, p->state.temp, p->flow_rate, p->state.time,
             p->user[0], name);
}

Hooking a DPM Scalar Update UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_SCALAR_UPDATE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Discrete Phase Model panel in FLUENT. See Section 8.5.9 for details on how to hook your DEFINE_DPM_SCALAR_UPDATE UDF to FLUENT.
4.6.10  DEFINE_DPM_SOURCE

Description

You can use the DEFINE_DPM_SOURCE macro to access particle source terms. The function allows access to the accumulated source terms for a particle in a given cell before they are added to the mass, momentum, and energy exchange terms for coupled DPM calculations.

Usage

Macro:    DEFINE_DPM_SOURCE (name, c, t, S, strength, p)

Argument types:  cell_t c
                    Thread *t
                    dpms_t *S
                    real strength
                    Tracked_Particle *p

Function returns:  void

There are six arguments to DEFINE_DPM_SOURCE: name, c, t, S, strength, and p. You will supply name, the name of the UDF. c, t, S, strength, and p are variables that are passed by the FLUENT solver to your UDF.

The passed variable c is an index that identifies the cell the particle is currently in. t is a pointer to the thread the particle is currently in. S is a pointer to the source structure dpms_t, which contains the source terms for the cell. strength is the particle number flow rate in particles/second (divided by the number of tries if stochastic tracking is used). p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties). The modified source terms, once computed by the function, will be stored in S.
Example

See Section 4.6.12 for an example of DEFINE_DPM_SOURCE usage.

Hooking a DPM Source Term UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_SOURCE is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Discrete Phase Model panel in FLUENT. See Section 8.5.10 for details on how to hook your DEFINE_DPM_SOURCE UDF to FLUENT.
4.6.11 DEFINE_DPM_SPRAY_COLLIDE

Description

You can use the DEFINE_DPM_SPRAY_COLLIDE macro to side-step the default FLUENT spray collision algorithm. When droplets collide they may bounce (in which case their velocity changes) or they may coalesce (in which case their velocity is changed, as well as their diameter and number in the DPM parcel). A spray collide UDF is called during droplet tracking after every droplet time step and requires that collision is enabled in the DPM panel.

Usage

Macro: DEFINE_DPM_SPRAY_COLLIDE (name, tp, p)

Argument Types: Tracked_Particle *tp
Particle *p

Function returns: void

There are three arguments to DEFINE_DPM_SPRAY_COLLIDE: name, tp, and p. You will supply name, the name of the UDF. tp and p are variables that are passed by the FLUENT solver to your UDF.

The input variables p are tp, the Tracked_Particle structure variable, and p, the Particle structure variable. FLUENT stores the particles p in a linked list. When collision is enabled, this linked list is ordered by the cell that the particle is currently in. As particles from this linked list are tracked, they are copied from the particle list into a Tracked_Particle structure.
Example

As a simple (and non-physical) example, the droplet diameters are assumed to relax to their initial diameter over a specified time $t_{\text{relax}}$. The droplet velocity is also assumed to relax to the mean velocity of all droplets in the cell over the same time scale.

```c
/**
 * Example UDF that demonstrates DEFINE_DPM_SPRAY_COLLIDE
 */
#include "udf.h"
#include "dpm.h"
#include "surf.h"

DEFINE_DPM_SPRAY_COLLIDE(udf_man_spray_collide, tp, p)
{
    /* non-physical collision UDF that relaxes the particle */
    /* velocity and diameter in a cell to the mean over the */
    /* specified time scale $t_{\text{relax}}$ */

    const real t_relax = 0.001; /* seconds */

    /* get the cell and Thread that the particle is currently in */
    cell_t c = RP_CELL(&(tp->cCell));
    Thread *t = RP_THREAD(&(tp->cCell));

    /* Particle index for looping over all particles in the cell */
    Particle *pi;

    /* loop over all particles in the cell to find their mass */
    /* weighted mean velocity and diameter */
    int i;
    real u_mean[3]={0.}, mass_mean=0.;
    real d_orig = tp->state.diam;
    real decay = 1. - exp(-t_relax);
    begin_particle_cell_loop(pi,c,t)
    {
        mass_mean += pi->state.mass;
        for(i=0;i<3;i++)
            u_mean[i] += pi->state.V[i]*pi->state.mass;
    }
    end_particle_cell_loop(pi,c,t)
```
/* relax particle velocity to the mean and diameter to the */
/* initial diameter over the relaxation time scale t_relax */
if( mass_mean > 0. )
{
    for(i=0;i<3;i++)
        u_mean[i] /= mass_mean;
    for(i=0;i<3;i++)
        tp->state.V[i] += decay*( u_mean[i] - tp->state.V[i] );
    tp->state.diam += decay*( P_INIT_DIAM(tp) - tp->state.diam );
    /* adjust the number in the droplet parcel to conserve mass */
    tp->number_in_parcel *= CUB( d_orig/tp->state.diam );
}

**Hooking a DPM Spray Collide UDF to FLUENT**

After the UDF that you have defined using `DEFINE_DPM_SPRAY_COLLIDE` is interpreted or compiled (see Chapter 7 for details), the name that you specified in the `DEFINE` macro argument will become visible in the User-Defined Function Hooks panel in FLUENT. See Section 8.5.11 for details on how to hook your `DEFINE_DPM_SPRAY_COLLIDE` UDF to FLUENT.
4.6.12 DEFINE_DPM_SWITCH

Description

You can use the DEFINE_DPM_SWITCH macro to modify the criteria for switching between laws. The function can be used to control the switching between the user-defined particle laws and the default particle laws, or between different user-defined or default particle laws.

Usage

Macro: DEFINE_DPM_SWITCH (name, p, ci)

Argument types: Tracked_Particle *p
int ci

Function returns: void

There are three arguments to DEFINE_DPM_SWITCH: name, p, and ci. You will supply name, the name of the UDF. p and ci are variables that are passed by the FLUENT solver to your UDF.

The passed variable p is a pointer to a Tracked_Particle structure that contains data relating to the particle being tracked. This pointer can be used as an argument to the macros defined in Section 5.6 to obtain information about particle properties (e.g., injection properties). ci is an integer that indicates if the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).

Example

The following is an example of a compiled UDF that uses DEFINE_DPM_SWITCH to switch between DPM laws using a criterion. The UDF switches to DPM_LAW_USER_1 which refers to condenshumidlaw since only one user law has been defined. The switching criterion is the local humidity which is computed in the domain using a DEFINE_ON_DEMAND function, which again calls the function myHumidity for every cell. In the case where the humidity is greater than 1, condensation is computed by applying a simple mass transfer calculation. Otherwise, one of FLUENT’s standard laws for Vaporization or Inert Heating are applied, depending on the particle mass. The UDF requires one UDML and needs a species called h2o to compute the local humidity.
/**********************************************************************
Concatenated UDFs for the Discrete Phase Model that includes a
usage of DPM_SWITCH
***********************************************************************/

#include "udf.h"
#include "dpm.h"
#include "surf.h" /* for macros: RP_Cell() & RP_Thread() */
#include "prop.h" /* for function: Saturation_Pressure() (of water) */

static int counter=0;
static real dpm_relax=1.0; /*dpm source relaxation */

real H2O_Saturation_Pressure(real T)
{
    real ratio, aTmTp;

    aTmTp = .01 * (T - 338.15);
    ratio = (647.286/T - 1.) *
    (-7.419242 + aTmTp*.29721 +
    aTmTp*-.1155286 +
    aTmTp*(8.685635e-3 +
    aTmTp*(1.094098e-3 +
    aTmTp*(-4.39993e-3 +
    aTmTp*(2.520658e-3 -
    aTmTp*5.218684e-4)))))
    return (22.089e6 * exp(MIN(ratio,35.)));
}

real myHumidity(cell_t c, Thread *t)
{
    int i;
    Material *m=THREAD_MATERIAL(t), *sp;
    real yi_h2o,mw_h2o;
    real r_mix=0.0;

    if(MATERIAL_TYPE(m)==MATERIAL_MIXTURE)
    {
        mixture_species_loop (m,sp,i)
        {
            r_mix += C_YI(c,t,i)/MATERIAL_PROP(sp,PROP_mwi);
if (0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"h2o") ||
(0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"H2O")))
{
  yi_h2o = C_YI(c,t,i);
  mw_h2o = MATERIAL_PROP(sp,PROP_mwi);
}
}

return ((ABS_P(C_P(c,t),op_pres) * yi_h2o / (mw_h2o * r_mix)) /
        H2O_Saturation_Pressure(C_T(c,t))) ;
}

#define CONDENS 1.0e-4

DEFINE_DPM_LAW(condenshumidlaw,p,coupled)
{
  real area;
  real mp_dot;
  cell_t c = P_CELL(p); /* Get Cell and Thread from */
  Thread *t = P_THREAD(p); /* Particle Structure using new macros*/

  area = 4.0* M_PI * (P_DIAM(p)*P_DIAM(p));

  /* Note This law only used if Humidity > 1.0 so mp_dot always positive*/
  mp_dot = CONDENS*sqrt(area)*(myHumidity(c,t)-1.0);

  if(mp_dot>0.0)
  {
    P_MASS(p) = P_MASS(p) + mp_dot*P_DT(p);
    P_DIAM(p) = pow(6.0*P_MASS(p)/(P_RHO(p)* M_PI), 1./3.);
    P_T(p)=C_T(c,t); /* Assume condensing particle is in thermal
                      equilibrium with fluid in cell */
  }
}

/* define macro that is not yet standard */

#define C_DPMS_ENERGY(c,t)C_STORAGE_R(c,t,SV_DPMS_ENERGY)
DEFINE_DPM_SOURCE(dpm_source, c, t, S, strength, p)
{
    real mp_dot;
    Material *sp = P_MATERIAL(p);

    /* mp_dot is the (positive) mass source to the continuous phase */
    /* (Difference in mass between entry and exit from cell) */
    /* multiplied by strength (Number of particles/s in stream) */

    mp_dot = (P_MASS0(p) - P_MASS(p)) * strength;

    C_DPMS_YI(c,t,0) += mp_dot*dpm_relax;
    C_DPMS_ENERGY(c,t) -= mp_dot*dpm_relax* MATERIAL_PROP(sp,PROP_Cp)*(C_T(c,t)-298.15);
    C_DPMS_ENERGY(c,t) -= mp_dot*dpm_relax* MATERIAL_PROP(sp,PROP_latent_heat);
}

#define UDM_RH 0
#define N_REQ_UDM 1
#define CONDENS_LIMIT 1.0e-10

DEFINE_DPM_SWITCH(dpm_switch,p,coupled)
{
    cell_t c = RP_CELL(&p->cCell);
    Thread *t = RP_THREAD(&p->cCell);

    if(C_UDMI(c,t,UDM_RH) > 1.0)
        P_CURRENT_LAW(p) = DPM_LAW_USER_1;
    else
    {
        if(P_MASS(p) < CONDENS_LIMIT)
            P_CURRENT_LAW(p) = DPM_LAW_INITIAL_INERT_HEATING;
        else
            P_CURRENT_LAW(p) = DPM_LAW_VAPORIZATION;
    }
}
DEFINE Macros

DEFINE_ADJUST(adj_relhum, domain)
{
    cell_t cell;
    Thread *thread;

    /* set dpm source underrealxation */
    dpm_relax = Domainvar_Get_Real(ROOT_DOMAIN_ID, "dpm/relax");

    if (sg_udm < N_REQ_UDM)
        Message("\nNot enough user defined memory allocated. %d required.\n", N_REQ_UDM);
    else
    {
        real humidity, min, max;

        min = 1e10;
        max = 0.0;

        thread_loop_c(thread, domain)
        {
            /* Check if thread is a Fluid thread and has UDMs set up on it */
            if (FLUID_THREAD_P(thread) && NNULLP(THREAD_STORAGE(thread, SV_UDM_I)))
            {
                begin_c_loop(cell, thread)
                humidity = myHumidity(cell, thread);
                min = MIN(min, humidity);
                max = MAX(max, humidity);
                C_UDMI(cell, thread, UDM_RH) = humidity;
                end_c_loop(cell, thread)
            }
        }
        Message("\nRelative Humidity set in udm-%d range: (%f, %f)\n", UDM_RH, min, max);
    } /* end if for enough UDSs and UDMs */
}

DEFINE_ON_DEMAND(set_relhum)
{
    adj_relhum(Get_Domain(1));
}
Hooking a DPM Switching UDF to FLUENT

After the UDF that you have defined using DEFINE_DPM_SWITCH is interpreted or compiled (see Chapter 7 for details), the name that you specified in the DEFINE macro argument will become visible in the Custom Laws panel in FLUENT. See Section 8.5.12 for details on how to hook your DEFINE_DPM_SWITCH UDF to FLUENT.
Chapter 5. Accessing Solver Data

This chapter provides predefined macros that you can use to access variables from the FLUENT solver.

- Section 5.1: Introduction
- Section 5.2: Grid Variables
- Section 5.3: Cell Variables
- Section 5.4: Face Variables
- Section 5.5: Multiphase Variables
- Section 5.6: DPM Variables
- Section 5.7: NO\textsubscript{x} Variables

5.1 Introduction

Access to solver variables is accomplished by hooking your UDF C function (once it is compiled and linked) to the solver through the FLUENT graphical user interface. Once the UDF is correctly hooked to the FLUENT solver, the solver’s data becomes available for the function to use whenever it is called. These data are automatically passed by the solver to your UDF as arguments to your function. Note that all solver data, regardless of whether they are passed to your UDF by the solver or returned to the solver by your UDF, are specified in SI units.

Fluent Inc. has provided you with a set of predefined functions that you can use to access data from the FLUENT solver. These functions are primarily implemented in the code as macros. The macros listed in this chapter are defined in header files such as mem.h, metric.h, and dpm.h. The udf.h file contains definitions for DEFINE macros, as well as #include directives for most of the solver access macro header files found in this chapter. Therefore, including udf.h in your source code file will also result in the inclusion of solver access .h files.
Some examples of solver data you can access using predefined macros are:

- solution variables and their derivatives (e.g., velocity, temperature, turbulence quantities)
- geometry variables (e.g., coordinates, areas, volumes)
- grid and node variables (e.g., node velocities)
- material property variables (e.g., density, viscosity, conductivity)
- discrete phase model variables

For all types of data except specific heat, the word “access” refers to reading and writing data. In the case of specific heat, however, data can be read but cannot be modified.

Access to solver variables is accomplished by hooking the C function (once it is compiled and linked) to the solver through the FLUENT user interface. Once the UDF is correctly hooked to the FLUENT solver, the solver’s data become available for the function to use whenever it is called. These data are automatically passed by the solver to your UDF as arguments to your function. Note that all solver data, regardless of whether they are passed to your UDF by the solver or returned to the solver by your UDF, are specified in SI units.

In the following sections, each macro is listed with its arguments, argument types, and returned value. Arguments belong to the following data types:

- cell_t c  
- face_t f  
- Thread *t  
- Thread **pt  
- int i  
- Node *node  

Arguments are either inputs to a function, or are outputs. Each macro returns a value, which either is output back to the solver (as an argument), or is available for assignment in your function.

For example, the macro C_T

```c
real temp;
temp = C_T(c,t);
```

has two arguments, cell identifier c and thread pointer t. These arguments are passed from the FLUENT solver to the function. C_T returns the real value of temperature, which can then be assigned to a variable in your UDF (temp in this example).
As another example, the macro `C_CENTROID`

\[ C_{\text{CENTROID}}(x,c,t); \]

has three arguments: \( x \), \( c \), and \( t \). In this case the cell identifier \( c \) and thread pointer \( t \) are input arguments, while the array \( x \) (the cell centroid) is output to the FLUENT solver as an argument.

## 5.2 Grid Variables

A grid in FLUENT is defined by the position of its nodes, and how the nodes are connected. The macros listed in this section can be used to return node position and connectivity information with respect to a grid.

### 5.2.1 Node Position Variables

The macros listed in Tables 5.2.1 can be used to return the real Cartesian coordinates of the cell node (at the cell corner) in SI units. The variables are available in both the segregated and the coupled solvers. Definitions for these macros can be found in the `metric.h` header file. The argument `Node *node` for each of the variables defines a node.

**Table 5.2.1: Macros for Node Coordinates (metric.h)**

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE_X(node)</td>
<td>Node *node</td>
<td>( x ) coordinate of node</td>
</tr>
<tr>
<td>NODE_Y(node)</td>
<td>Node *node</td>
<td>( y ) coordinate of node</td>
</tr>
<tr>
<td>NODE_Z(node)</td>
<td>Node *node</td>
<td>( z ) coordinate of node</td>
</tr>
</tbody>
</table>

### 5.2.2 Connectivity Variables

The macros listed in this section can be used to return real connectivity variables in SI units. The variables are available in both the segregated and the coupled solvers. Definitions for these node macros can be found in the referenced header file (e.g., `mem.h`).
Accessing Solver Data

Number of Nodes and Faces

The macros \texttt{C\_NNODES} and \texttt{C\_NFACES} shown in Table 5.2.2 return the integer number of nodes or faces, respectively, for a given cell. \texttt{F\_NNODES} returns the integer number of nodes associated with a face.

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{C_NNODES(c,t)}</td>
<td>\texttt{cell_t c, Thread *t}</td>
<td>number of nodes in a cell</td>
</tr>
<tr>
<td>\texttt{C_NFACES(c,t)}</td>
<td>\texttt{cell_t c, Thread *t}</td>
<td>number of faces in a cell</td>
</tr>
<tr>
<td>\texttt{F_NNODES(f,t)}</td>
<td>\texttt{face_t f, Thread *t}</td>
<td>number of nodes in a face</td>
</tr>
</tbody>
</table>

Cell and Face Centroids

The macros listed in Table 5.2.3 can be used to obtain the real centroid of a cell or face. \texttt{C\_CENTROID} finds the coordinate position of the centroid of the cell \texttt{c} and stores the coordinates in the \texttt{x} array. \texttt{F\_CENTROID} finds the coordinate position of the centroid of the face \texttt{f} and stores the coordinates in the \texttt{x} array. Note that the array \texttt{x} can be a one-, two-, or three-dimensional array.

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{C_CENTROID(x,c,t)}</td>
<td>\texttt{real x[ND_ND], cell_t c, Thread *t}</td>
<td>\texttt{x (cell centroid)}</td>
</tr>
<tr>
<td>\texttt{F_CENTROID(x,f,t)}</td>
<td>\texttt{real x[ND_ND], face_t f, Thread *t}</td>
<td>\texttt{x (face centroid)}</td>
</tr>
</tbody>
</table>
5.2 Grid Variables

**Face Area Vector (F_AREA)**

*F_AREA* can be used to return the real face area vector (or ‘face area normal’) of a given face *f* in a face thread *tf*.

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_AREA(A,f,tf)</td>
<td>A[ND,ND], face_t f, Thread *tf</td>
<td>A (area vector)</td>
</tr>
</tbody>
</table>

By convention in *FLUENT*, boundary face area normals always point out of the domain. *FLUENT* determines the direction of the face area normals for interior faces by applying the right hand rule to the nodes on a face, in order of increasing node number. This is shown in Figure 5.2.1.

*FLUENT* assigns adjacent cells to an interior face (c0 and c1) according to the following convention: the cell *out* of which a face area normal is pointing is designated as cell c0, while the cell *in* to which a face area normal is pointing is cell c1 (Figure 5.2.1). In other words, face area normals always point from cell c0 to cell c1.

---

Figure 5.2.1: *FLUENT* Determination of Face Area Normal Direction: 2D Face

![Diagram showing face area normal direction](image)
Cell Volume

The macros listed in Table 5.2.5 can be used to obtain the real cell volume for 2D, 3D, and axisymmetric simulations.

Table 5.2.5: Macros for Cell Volume (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_VOLUME(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>cell volume for 2D or 3D, cell volume/2π for axisymmetric</td>
</tr>
</tbody>
</table>

Cell-to-Cell and Cell-to-Face Centroid Macros

FLUENT provides macros that allow the vectors connecting cell centroids and the vectors connecting cell and face centroids to be readily defined. These macros return information that is helpful in evaluating face values of scalars which are generally not stored, as well as the diffusive flux of scalars across cell boundaries. The geometry and gradients involved with these macros are summarized in Figure 5.2.2 below.

To better understand the parameters that are returned by these macros, it is best to consider how the aforementioned calculations are evaluated. Assuming that the gradient of a scalar is available, the face value of a scalar, φ, can be approximated by

\[
\phi_f = \phi_0 + \nabla \phi \cdot \vec{dr}
\]  
(5.2-1)

where \(\vec{dr}\) is the vector that connects the cell centroid with the face centroid. The gradient in this case is evaluated at the cell centroid where \(\phi_0\) is also stored.

The diffusive flux, \(D_f\), across a face, \(f\), of a scalar \(\phi\) is given by,

\[
D_f = \Gamma_f \nabla \phi \cdot \vec{A}
\]  
(5.2-2)

where \(\Gamma_f\) is the diffusion coefficient at the face. In FLUENT’s unstructured solver, the gradient along the face normal direction may be approximated by evaluating gradients along the directions that connect cell centroids and along a direction confined within the plane of the face. Given this, \(D_f\) may be approximated as,

\[
D_f = \Gamma_f \left( \frac{\phi_1 - \phi_0}{ds} \frac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e_s}}} + \Gamma_f \left( \nabla \phi \cdot \vec{A} - \nabla \phi \cdot \vec{e_s} \frac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e_s}}} \right) \right)
\]  
(5.2-3)
where the first term on the right hand side represents the primary gradient directed along
the vector $\vec{e}_s$ and the second term represents the ‘cross’ diffusion term. In this equation,
$A$ is the area normal vector of face $f$ directed from cell $c_0$ to $c_1$, $ds$ is the distance
between the cell centroids, and $\vec{e}_s$ is the unit normal vector in this direction. This is
shown in Figure 5.2.2 below.

![Figure 5.2.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions](image)

The `INTERIOR_FACE_GEOMETRY` and `BOUNDARY_FACE_GEOMETRY` macros can be called to
return some of the terms needed to evaluate Equations 5.2-1 and 5.2-3.

**INTERIOR_FACE_GEOMETRY**

`INTERIOR_FACE_GEOMETRY(f,tf,A,ds,es,A_by_es,dr0,dr1)` returns for the face, $f$, in
face thread $tf$, the following variables:

- $A$ (ND_ND): the area normal vector
- $ds$: the distance between the cell centroids
- $es$ (ND_ND): the unit normal vector in the direction from cell $c_0$ to $c_1$
- $A_{by\_es}$: the value $\frac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e}_s}$
- $dr0$ (ND_ND): the vector that connects the centroid of cell $c_0$ to the face centroid
- $dr1$ (ND_ND): the vector that connects the centroid of cell $c_1$ to the face centroid

Note that the `INTERIOR_FACE_GEOMETRY` macro is defined in the `sg.h` header file. Since
`sg.h` is not included in the `udf.h` file, you will need to add the `#include` directive to
your UDF.
BOUNDARY_FACE_GEOMETRY

BOUNDARY_FACE_GEOMETRY(f, tf, A, ds, es, A_by_es, dr0) returns for the face, f, in face thread tf, the following variables:

- **A [ND_ND]** the area normal vector
- **ds** the distance between the cell centroid and the face centroid
- **es [ND_ND]** the unit normal vector in the direction from centroid of cell c0 to the face centroid
- **A_by_es** the value $\frac{\vec{A} \cdot \vec{A}}{\epsilon}$
- **dr0 [ND_ND]** the vector that connects the centroid of cell c0 to the face centroid

The BOUNDARY_FACE_GEOMETRY macro is also defined in the sg.h header file, and is not included in the udf.h file. You will need to include sg.h in your UDF.

BOUNDARY_FACE_THREAD_P(tf)

This macro expands to a function that returns TRUE if Thread *tf is a boundary face thread.

Adjacent Cell Index (F_C0, F_C1)

There are two macros, F_C0(f, tf) and F_C1(f, tf), that can be used to identify cells that are adjacent to a given face thread tf. F_C0 returns the index of a face’s neighboring c0 cell, while F_C1 returns the cell index for c1 (if it exists).

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_C0(f, tf)</td>
<td>face_t f, Thread *tf</td>
<td>cell_t c for cell c0</td>
</tr>
<tr>
<td>F_C1(f, tf)</td>
<td>face_t f, Thread *tf</td>
<td>cell_t c for cell c1</td>
</tr>
</tbody>
</table>
Adjacent Cell Thread (THREAD_T0, THREAD_T1)

There are two macros, THREAD_T0(tf) and THREAD_T1(tf), that can be used to identify cell threads that are adjacent to a given face f in a face thread tf. THREAD_T0 expands to a function that returns the cell thread of a given face's adjacent cell c0, and THREAD_T1(tf) returns the cell thread for c1 (if it exists).

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>THREAD_T0(tf)</td>
<td>Thread *tf</td>
<td>cell thread pointer for cell c0</td>
</tr>
<tr>
<td>THREAD_T1(tf)</td>
<td>Thread *tf</td>
<td>cell thread pointer for cell c1</td>
</tr>
</tbody>
</table>

C_FACETHREAD(c,t,i)

This macro returns the Thread *t of the face_t f that is returned by C_FACE. Note that the integer index i can also be accessed by c_face_loop.

C_FACE(c,t,i)

This macro returns the face face_t f for the given cell_t c and Thread *t. Note that the integer index i can also be accessed by c_face_loop.

5.3 Cell Variables

Dependent cell data that are stored in a FLUENT solver can be accessed using macros listed in this section. The macros can be used to return real variables in SI units. The cell variables are available in both the segregated and the coupled solvers. Definitions for these cell variable macros can be found in the referenced header file (e.g., mem.h).

Flow Variable Macros

Gradient (G) and Reconstruction Gradient (RG) Vector Macros

You can access gradient and reconstruction gradient vectors (and components) for many of the cell variables listed in Table 5.3.1. FLUENT calculates the gradient of flow in a cell (based on the divergence theory) and stores this value in the variable identified by the suffix _G. For example cell temperature is stored in the variable C.T, and the temperature gradient of the cell is stored in C.T.G. The gradients stored in variables with the _G suffix are non-limited values and if used to reconstruct values within the cell (at faces, for example), may potentially result in values that are higher (or lower) than values in
Table 5.3.1: Macros for Cell Flow Variables (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_R(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>density</td>
</tr>
<tr>
<td>C_P(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>pressure</td>
</tr>
<tr>
<td>C_U(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>u velocity</td>
</tr>
<tr>
<td>C_V(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>v velocity</td>
</tr>
<tr>
<td>C_W(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>w velocity</td>
</tr>
<tr>
<td>C_T(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>temperature</td>
</tr>
<tr>
<td>C_H(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>enthalpy</td>
</tr>
<tr>
<td>C_K(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turb. kinetic energy</td>
</tr>
<tr>
<td>C_NUT(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent viscosity</td>
</tr>
<tr>
<td>C_D(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turb. kinetic energy</td>
</tr>
<tr>
<td>C_O(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>dissipation rate</td>
</tr>
<tr>
<td>C_YI(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>species mass fraction</td>
</tr>
</tbody>
</table>

the surrounding cells. Therefore, if your UDF needs to compute face values from cell gradients, you should use the reconstruction gradient (RG) values instead of non-limited gradient (G) values. Reconstruction gradient variables are identified by the suffix _RG, and use the limiting method that you have activated in your FLUENT model to limit the cell gradient values.

**Gradient (G) Vector Macros**

Table 5.3.2 shows a list of cell gradient vector macros. Note that gradient variables are available only when the equation for that variable is being solved. For example, if you are defining a source term for energy, your UDF can access the cell temperature gradient (using C_T_G), but it can not get access to the x-velocity gradient (using C_U_G). The reason for this is that the solver continually removes data from memory that it doesn’t need. In order to retain the gradient data (when you want to set up user-defined scalar transport equations, for example), you can prevent the solver from freeing up memory by issuing the text command solve/set/expert and then answering yes to the question *Keep temporary solver memory from being freed?*. Note that when you do this, all of the gradient data is retained, but the calculation requires more memory to run.

You can access a component of a gradient vector by specifying it as an argument in the gradient vector call (0 for the x component; 1 for y; and 2 for z). For example,

```c
C_T_G(c,t)[0];  /* returns the x-component of the cell temperature */  
/* gradient vector */
```
returns the $x$ component of the temperature gradient vector.

**Reconstruction Gradient (RG) Vector Macros**

Table 5.3.3 shows a list of cell reconstruction gradient vector macros. Like gradient variables, RG variables are available only when the equation for that variable is being solved. As in the case of gradient variables, you can retain all of the reconstruction gradient data by issuing the text command `solve/set/expert` and then answering `yes` to the question `Keep temporary solver memory from being freed?`. Note that when you do this, the reconstruction gradient data is retained, but the calculation requires more memory to run.

You can access a component of a reconstruction gradient vector by specifying it as an argument in the reconstruction gradient vector call ($0$ for the $x$ component; $1$ for $y$; and $2$ for $z$). For example,

```plaintext
C_T_RG(c,t)[0];  /* returns the x-component of the cell temperature */  /* reconstruction gradient vector */
```

returns the $x$ component of the temperature reconstruction gradient vector.

**Previous Time Step Macros**

The `_M1` suffix can be applied to some of the cell variable macros in Table 5.3.1 to allow access to the value of the variable at the previous time step (i.e., $t - \Delta t$). These data may be useful in unsteady simulations. For example,

```plaintext
C_T_M1(c,t);
```

returns the value of the cell temperature at the previous time step. Previous time step macros are shown in Table 5.3.4.

Note that data from `C_T_M1` is available *only* if user-defined scalars are used.

The `_M2` suffix can be applied to some of the cell variable macros in Table 5.3.1 to allow access to the value of the variable at the time step before the previous one (i.e., $t - 2\Delta t$). These data may be useful in unsteady simulations. For example,

```plaintext
C_T_M2(c,t);
```

returns the value of the cell temperature at the time step before the previous one. Two previous time step macros are shown in Table 5.3.5.

Note that data from `C_T_M2` is available *only* if user-defined scalars are used.
# Accessing Solver Data

## Table 5.3.2: Macros for Cell Gradients (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_R.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>density gradient vector</td>
</tr>
<tr>
<td>C_R.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>density gradient component</td>
</tr>
<tr>
<td>C_P.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>pressure gradient vector</td>
</tr>
<tr>
<td>C_P.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>pressure gradient component</td>
</tr>
<tr>
<td>C_U.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity gradient vector</td>
</tr>
<tr>
<td>C_U.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>velocity gradient component</td>
</tr>
<tr>
<td>C_V.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity gradient vector</td>
</tr>
<tr>
<td>C_V.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>velocity gradient component</td>
</tr>
<tr>
<td>C_W.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity gradient vector</td>
</tr>
<tr>
<td>C_W.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>velocity gradient component</td>
</tr>
<tr>
<td>C_T.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>temperature gradient vector</td>
</tr>
<tr>
<td>C_T.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>temperature gradient component</td>
</tr>
<tr>
<td>C_H.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>enthalpy gradient vector</td>
</tr>
<tr>
<td>C_H.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>enthalpy gradient component</td>
</tr>
<tr>
<td>C_NUT.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent viscosity for Spalart-Allmaras gradient vector</td>
</tr>
<tr>
<td>C_NUT.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>turbulent viscosity for Spalart-Allmaras gradient component</td>
</tr>
<tr>
<td>C_K.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent kinetic energy gradient vector</td>
</tr>
<tr>
<td>C_K.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>turbulent kinetic energy gradient component</td>
</tr>
<tr>
<td>C_D.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent kinetic energy dissipation rate gradient vector</td>
</tr>
<tr>
<td>C_D.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>turbulent kinetic energy dissipation rate gradient component</td>
</tr>
<tr>
<td>C_O.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>specific dissipation rate gradient vector</td>
</tr>
<tr>
<td>C_O.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>specific dissipation rate gradient component</td>
</tr>
<tr>
<td>C_YI.G(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>species mass fraction gradient vector</td>
</tr>
<tr>
<td>C_YI.G(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>species mass fraction gradient component</td>
</tr>
</tbody>
</table>
Table 5.3.3: Macros for Cell Reconstruction Gradients (RG) (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_R_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>density RG vector</td>
</tr>
<tr>
<td>C_R_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>density RG component</td>
</tr>
<tr>
<td>C_P_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>pressure RG vector</td>
</tr>
<tr>
<td>C_P_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>pressure RG component</td>
</tr>
<tr>
<td>C_U_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity RG vector</td>
</tr>
<tr>
<td>C_U_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>velocity RG component</td>
</tr>
<tr>
<td>C_V_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity RG vector</td>
</tr>
<tr>
<td>C_V_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>velocity RG component</td>
</tr>
<tr>
<td>C_W_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity RG vector</td>
</tr>
<tr>
<td>C_W_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>velocity RG component</td>
</tr>
<tr>
<td>C_T_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>temperature RG vector</td>
</tr>
<tr>
<td>C_T_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>temperature RG component</td>
</tr>
<tr>
<td>C_H_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>enthalpy RG vector</td>
</tr>
<tr>
<td>C_H_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>enthalpy RG component</td>
</tr>
<tr>
<td>C_NUT_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent viscosity for Spalart-Allmaras RG vector</td>
</tr>
<tr>
<td>C_NUT_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>turbulent viscosity for Spalart-Allmaras RG component</td>
</tr>
<tr>
<td>C_KRG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent kinetic energy RG vector</td>
</tr>
<tr>
<td>C_KRG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>turbulent kinetic energy RG component</td>
</tr>
<tr>
<td>C_DRG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent kinetic energy dissipation rate RG vector</td>
</tr>
<tr>
<td>C_DRG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>turbulent kinetic energy dissipation rate RG component</td>
</tr>
<tr>
<td>C_YI_RG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>species mass fraction RG vector</td>
</tr>
<tr>
<td>C_YI_RG(c,t)[i]</td>
<td>cell_t c, Thread *t, int i</td>
<td>species mass fraction RG component</td>
</tr>
</tbody>
</table>
Table 5.3.4: Macros for Cell Time Level 1 (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_R_M1(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>density, previous time step</td>
</tr>
<tr>
<td>C_P_M1(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>pressure, previous time step</td>
</tr>
<tr>
<td>C_U_M1(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity, previous time step</td>
</tr>
<tr>
<td>C_V_M1(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity, previous time step</td>
</tr>
<tr>
<td>C_W_M1(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity, previous time step</td>
</tr>
<tr>
<td>C_T_M1(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>temperature, previous time step</td>
</tr>
<tr>
<td>C_YI_M1(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>species mass fraction, previous time step</td>
</tr>
</tbody>
</table>

Table 5.3.5: Macros for Cell Time Level 2 (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_R_M2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>density, previous time step</td>
</tr>
<tr>
<td>C_P_M2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>pressure, previous time step</td>
</tr>
<tr>
<td>C_U_M2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity, previous time step</td>
</tr>
<tr>
<td>C_V_M2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity, previous time step</td>
</tr>
<tr>
<td>C_W_M2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity, previous time step</td>
</tr>
<tr>
<td>C_T_M2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>temperature, previous time step</td>
</tr>
<tr>
<td>C_YI_M2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>species mass fraction, previous time step</td>
</tr>
</tbody>
</table>
5.3 Cell Variables

Derivative Macros

The macros listed in Table 5.3.6 can be used to return real velocity derivative variables in SI units. The variables are available in both the segregated and the coupled solvers. Definitions for these macros can be found in the mem.h header file.

Table 5.3.6: Macros for Cell Velocity Derivatives (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_STRAIN_RATE_MAG(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>strain rate magnitude</td>
</tr>
<tr>
<td>C_DUDX(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DUDY(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DUDZ(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DVDX(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DVDY(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DVDZ(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DWDX(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DWDY(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
<tr>
<td>C_DWDZ(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>velocity derivative</td>
</tr>
</tbody>
</table>

Material Property Macros

The macros listed in Tables 5.3.7–5.3.9 can be used to return real material property variables in SI units. The variables are available in both the segregated and the coupled solvers. Definitions for these macros can be found in the referenced header file (e.g., mem.h).

Reynolds Stress Model Macros

The macros listed in Table 5.3.10 can be used to return real variables for the Reynolds stress turbulence model in SI units. The variables are available in both the segregated and the coupled solvers. Definitions for these macros can be found in the metric.h header file.

User-Defined Scalar Variables for Cells (C_UDSI)

You can use C_UDSI when you want to access cell variables that are computed for user-defined scalar transport equations. Macros for accessing UDS cell variables are listed in Table 5.3.11. Some examples of usage for these macros include defining non-constant source terms for UDS transport equations and initializing equations. An example of C_UDSI usage is shown below.

© Fluent Inc. January 22, 2003 5-15
### Table 5.3.7: Macros for Diffusion Coefficients (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_MU_L(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>laminar viscosity</td>
</tr>
<tr>
<td>C_MU_T(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent viscosity</td>
</tr>
<tr>
<td>C_MU_EFF(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>effective viscosity</td>
</tr>
<tr>
<td>C_K_L(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>thermal conductivity</td>
</tr>
<tr>
<td>C_K_T(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent thermal conductivity</td>
</tr>
<tr>
<td>C_K_EFF(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>effective thermal conductivity</td>
</tr>
<tr>
<td>C_DIFF_L(c,t,i,j)</td>
<td>cell_t c, Thread *t, int i, int j</td>
<td>laminar species diffusivity</td>
</tr>
<tr>
<td>C_DIFF_EFF(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>effective species diffusivity</td>
</tr>
</tbody>
</table>

### Table 5.3.8: Macros for Thermodynamic Properties (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_CP(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>specific heat</td>
</tr>
<tr>
<td>C_RGAS(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>gas constant</td>
</tr>
<tr>
<td>C_ABS_COEFF(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>absorption coefficient</td>
</tr>
<tr>
<td>C_SCAT_COEFF(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>scattering coefficient</td>
</tr>
<tr>
<td>C_NUT(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>turbulent viscosity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for Spalart-Allmaras</td>
</tr>
</tbody>
</table>
### 5.3 Cell Variables

#### Table 5.3.9: Additional Material Property Macros (\texttt{sg\_mem.h})

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_FMEAN(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>primary mean mixture fraction</td>
</tr>
<tr>
<td>C_FMEAN2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>secondary mean mixture fraction</td>
</tr>
<tr>
<td>C_FVAR(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>primary mixture fraction variance</td>
</tr>
<tr>
<td>C_FVAR2(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>secondary mixture fraction variance</td>
</tr>
<tr>
<td>C_PREMIXC(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>reaction progress variable</td>
</tr>
<tr>
<td>C_LAM_FLAME_SPEED(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>laminar flame speed</td>
</tr>
<tr>
<td>C_CRITICAL_STRAIN_ RATE(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>critical strain rate</td>
</tr>
<tr>
<td>C_POLLUT(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>(i)th pollutant species mass fraction</td>
</tr>
</tbody>
</table>

#### Table 5.3.10: Macros for Reynolds Stress Model Variables (\texttt{metric.h})

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_RUU(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>(uu) Reynolds stress</td>
</tr>
<tr>
<td>C_RVV(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>(vv) Reynolds stress</td>
</tr>
<tr>
<td>C_RWW(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>(ww) Reynolds stress</td>
</tr>
<tr>
<td>C_RUV(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>(uw) Reynolds stress</td>
</tr>
<tr>
<td>C_RVW(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>(vw) Reynolds stress</td>
</tr>
<tr>
<td>C_RUW(c,t)</td>
<td>cell_t c, Thread *t</td>
<td>(uw) Reynolds stress</td>
</tr>
</tbody>
</table>
Accessing Solver Data

Table 5.3.11: User-Defined Scalar Variables for Cells (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_UDSI(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>user-defined scalar</td>
</tr>
<tr>
<td>C_UDSI_G(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>user-defined scalar gradient</td>
</tr>
<tr>
<td>C_UDSI_M(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>user-defined scalar previous time step</td>
</tr>
<tr>
<td>C_UDSI_DIFF(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>user-defined scalar diffusivity</td>
</tr>
</tbody>
</table>

User-Defined Memory Variables for Cells (C_UDMI)

You can use C_UDMI when you want to store cell variables that are computed for user-defined scalar transport equations. C_UDMI can be used to allocate up to 500 memory locations in order to store and retrieve the values of cell field variables computed by UDFs. These stored values can then be used for postprocessing, for example, or by other UDFs. An example of C_UDMI usage is shown below.

Table 5.3.12: User-Defined Memory Variables for Cells (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_UDMI(c,t,i)</td>
<td>cell_t c, Thread *t, int i</td>
<td>stores data in user-defined cell memory location for given index</td>
</tr>
</tbody>
</table>

There are three arguments to C_UDMI: c, thread, and i. c is the cell identifier, thread is a pointer to the cell thread, and i is an integer index that identifies the memory location where data is to be stored. An index i of 0 corresponds to user-defined memory location 0 (or udm-0).

Before you can use C_UDMI to store variables in memory, you will first need to allocate memory location(s) in the User-Defined Memory panel in FLUENT. (See Section 8.1.6 for more details.)

! Define —→ User-Defined —→ Memory...

If you try using C_UDMI before you have allocated memory, then an error will result.

A variable will be created for every user-defined memory location that you allocate in the graphical user-interface. For example, if you specify 2 memory locations, then two
variables, \texttt{udm-0} and \texttt{udm-1}, will be available for data storage. (These names will appear in postprocessing panels.)

**Example - UDM and UDS variable Usage**

Below is an example of a UDF that begins by filling \texttt{uds-0} with the cell temperature raised to the fourth power (using \texttt{C.UDSI}). Then it fills \texttt{udm-0} with the gradient of this cell temperature, \texttt{C.UDSI.G} (using \texttt{C.UDMI}) that will be used later for post-processing.

```c
UDF used to calculate gradient for post-processing

User defined scaler (UDS) and user defined memory (UDM) are used. For any UDS, Fluent will automatically calculate the gradient. So, we need to pass the variable to UDS, and fluent will calculate the gradient for you. UDM is used to save the results.

Steps to take to make it work:
1. Build UDF library
2. Read in the converged case and data
3. Hook the UDF (Define->User Defined->Function->Compiled)
4. Hook adjust function (Define->User Defined->Function Hooks->Adjust Function)
5. Define UDM (Define->User Defined->Memory 1)
6. Define UDS (Define->User Defined->Scalars 1)
7. Define BC for UDS (the same as the function for which you want gradient)
8. Change UR for UDS to 1e-6 (Solve->Controls->Solution)
9. Turn off all equations except UDS (Solve->Controls->Solution)
10. Do A FEW iterations
11. Execute store_gradient (Define->User Defined->Execute On Demand)

NOTES on item 7 8 and 9
step 7 is VERY IMPORTANT, otherwise the gradient close to the boundary will not be correct.
step 8 is VERY IMPORTANT, otherwise the solution will change. Fluent will try to solve an equation for UDS; we want to force it to be unchanged.
step 9 will save you time as the solution is already converged.
step 10 Three is determined to be the minimum.
```

© Fluent Inc. January 22, 2003
# include "udf.h"
# define domain_ID 2

DEFINE_ADJUST(adjust_gradient, domain)
{
    Thread *t;
    cell_t c;
    face_t f;

    domain = Get_Domain(domain_ID);

    /* Fill UDS with the variable. */
    thread_loop_c (t,domain)
    {
        begin_c_loop (c,t)
        {
            C_UDSI(c,t,0) = C_VOF(c,t);
        }
        end_c_loop (c,t)
    }
}

DEFINE_ON_DEMAND(store_gradient)
{
    Domain *domain;
    cell_t c;
    Thread *t;

    domain = Get_Domain(1);

    /* Fill the UDM with magnitude of gradient. */
    thread_loop_c (t,domain)
    {
        begin_c_loop (c,t)
        {
            C_UDMI(c,t,0) = NV_MAG(C_UDSI_G(c,t,0));
        }
        end_c_loop (c,t)
    }
}
5.4 Face Variables

The macros listed in Table 5.4.1–5.4.2 can be used to return real face variables in SI units. Note that these variables are available only in the segregated solver. In addition, quantities that are returned are available only if the corresponding physical model is active. For example, species mass fraction is available only if species transport has been enabled in the Species Model panel in FLUENT. Definitions for these macros can be found in the referenced header files (e.g., mem.h).

Flow Variable Macros at a Boundary Face

The macros listed in Table 5.4.1 access flow variables at a boundary face.

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_R(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>density</td>
</tr>
<tr>
<td>F_U(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>u velocity</td>
</tr>
<tr>
<td>F_V(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>v velocity</td>
</tr>
<tr>
<td>F_W(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>w velocity</td>
</tr>
<tr>
<td>F_T(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>temperature</td>
</tr>
<tr>
<td>F_H(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>enthalpy</td>
</tr>
<tr>
<td>F_K(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>turbulent kinetic energy</td>
</tr>
<tr>
<td>F_D(f,tf)</td>
<td>face_t f, Thread *tf,</td>
<td>turbulent kinetic energy</td>
</tr>
<tr>
<td>F_YI(f,tf,i)</td>
<td>face_t f, Thread *tf, int i</td>
<td>species mass fraction</td>
</tr>
</tbody>
</table>

Flow Variable Macros at Interior and Boundary Faces

The macros listed in Table 5.4.1 access flow variables at interior faces and boundary faces.

F_FLUX can be used to return the real scalar mass flow rate through a given face f in a face thread tf. The sign of F_FLUX that is computed by the FLUENT solver is positive if the flow direction is the same as the face area normal direction (as determined by F_AREA - see Section 5.2.2), and is negative if the flow direction and the face area normal directions are opposite. In other words, the flux is positive if the flow is out of the domain, and is negative if the flow is in to the domain.

Note that the sign of the flux that is computed by the solver is opposite to that which is reported in the FLUENT graphical user-interface (e.g., Reports → Summary...).
Table 5.4.2: Macros for Face Flow Variables - Interior and Boundary Faces (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_P(f,tf)</td>
<td>face_t f, Thread *tf</td>
<td>pressure</td>
</tr>
<tr>
<td>F_FLUX(f,tf)</td>
<td>face_t f, Thread *tf</td>
<td>mass flow rate through a face</td>
</tr>
</tbody>
</table>

User-Defined Scalar Variable for Faces (F_UDSI)

You can use F_UDSI when you want to access face variables that are computed for user-defined scalar transport equations.

Table 5.4.3: User-Defined Scalar Variables for Faces (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_UDSI(f,t,i)</td>
<td>face_t f, Thread *t, int i</td>
<td>user-defined scalar</td>
</tr>
</tbody>
</table>

User-Defined Memory Variable for Faces (F_UDMI)

You can use F_UDMI when you want to store face variables that are computed for user-defined scalar transport equations. F_UDMI can be used to allocate up to 500 memory locations in order to store and retrieve the values of face field variables computed by UDFs. These stored values can then be used for postprocessing, for example, or by other UDFs.

There are three arguments to F_UDMI: c, thread, and i. c is the cell identifier, thread is a pointer to the cell thread, and i is an integer index that identifies the memory location where data is to be stored. An index i of 0 corresponds to user-defined memory location 0 (or udm-0).

Before you can use F_UDMI to store variables in memory, you will first need to allocate memory location(s) in the User-Defined Memory panel in FLUENT. (See Section 8.1.6 for more details.)

Define → User-Defined → Memory...

! If you try using F_UDMI before you have allocated memory, then an error will result.
Table 5.4.4: User-Defined Memory Variable for Faces (mem.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_UDMI(f,t,i)</td>
<td>face_t f, Thread *t, int i</td>
<td>stores data in user-defined face memory location for given index</td>
</tr>
</tbody>
</table>

Example

```c
/* Compute face temperature and store in user-defined memory */
begin_f_loop(f,t)
{
    temp = F_T(f,t);
    F_UDMI(f,t,0) = (temp - tmin) / (tmax-tmin);
}
end_f_loop(f,t)
```

5.5 Multiphase Variables

The macros listed in Table 5.5.1 can be used to return real variables associated with the general multiphase models in SI units. The variables are available in both the segregated and the coupled solvers. Definitions for these macros can be found in sg_mphase.h, which is included in udf.h.

Table 5.5.1: Macros for Multiphase Variables (sg_mphase.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>C_VOF(c,pt[0])</td>
<td>cell_t c, Thread **pt</td>
<td>volume fraction for primary phase</td>
</tr>
<tr>
<td>C_VOF(c,pt[n])</td>
<td>cell_t c, Thread **pt</td>
<td>volume fraction for nth secondary phase</td>
</tr>
</tbody>
</table>
5.6 DPM Variables

The macros listed in Tables 5.6.1–5.6.6 can be used to return real variables associated with the Discrete Phase Model (DPM), in SI units. The variables are available in both the segregated and the coupled solvers. The macros are defined in the dpm.h header file, which is included in udf.h.

The variable p indicates a pointer to the Tracked_Particle structure (Tracked_Particle *p). Tracked_Particle *p gives you the value for the particle at the current position.

Table 5.6.1: Macros for Particles at Current Position (dpm.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_POS(p)[i]</td>
<td>Tracked_Particle *p int i</td>
<td>position i=0,1,2</td>
</tr>
<tr>
<td>P_VEL(p)[i]</td>
<td>Tracked_Particle *p int i</td>
<td>velocity i=0,1,2</td>
</tr>
<tr>
<td>P_DIAM(p)</td>
<td>Tracked_Particle *p</td>
<td>diameter</td>
</tr>
<tr>
<td>P_T(p)</td>
<td>Tracked_Particle *p</td>
<td>temperature</td>
</tr>
<tr>
<td>P_RHO(p)</td>
<td>Tracked_Particle *p</td>
<td>density</td>
</tr>
<tr>
<td>P_MASS(p)</td>
<td>Tracked_Particle *p</td>
<td>mass</td>
</tr>
<tr>
<td>P_TIME(p)</td>
<td>Tracked_Particle *p</td>
<td>current particle time</td>
</tr>
<tr>
<td>P_DT(p)</td>
<td>Tracked_Particle *p</td>
<td>time step</td>
</tr>
<tr>
<td>P_FLOW_RATE(p)</td>
<td>Tracked_Particle *p</td>
<td>flow rate of particles in a stream in kg/s (see below for details)</td>
</tr>
<tr>
<td>P_LF(p)</td>
<td>Tracked_Particle *p</td>
<td>liquid fraction (wet combusting particles only)</td>
</tr>
<tr>
<td>P_VFF(p)</td>
<td>Tracked_Particle *p</td>
<td>volatile fraction (combusting particles only)</td>
</tr>
</tbody>
</table>

P_FLOW_RATE(p)

Each particle in a steady flow calculation represents a “stream” of many particles that follow the same path. The number of particles in this stream that passes a particular point in a second is the “strength” of the stream. P_FLOW_RATE returns the strength multiplied by P_MASS(p) at the current particle position.
### 5.6 DPM Variables

#### Table 5.6.2: Macros for Particles at Entry to Current Cell \( (\text{dpm.h}) \)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_POS0(p)[i]</td>
<td>Tracked_Particle *p int i</td>
<td>position ( i=0,1,2 )</td>
</tr>
<tr>
<td>P_VEL0(p)[i]</td>
<td>Tracked_Particle *p int i</td>
<td>velocity ( i=0,1,2 )</td>
</tr>
<tr>
<td>P_DIAM0(p)</td>
<td>Tracked_Particle *p</td>
<td>diameter</td>
</tr>
<tr>
<td>P_T0(p)</td>
<td>Tracked_Particle *p</td>
<td>temperature</td>
</tr>
<tr>
<td>P_RH00(p)</td>
<td>Tracked_Particle *p</td>
<td>density</td>
</tr>
<tr>
<td>P_MASS0(p)</td>
<td>Tracked_Particle *p</td>
<td>mass</td>
</tr>
<tr>
<td>P_TIME0(p)</td>
<td>Tracked_Particle *p</td>
<td>particle time at entry</td>
</tr>
<tr>
<td>P_LF0(p)</td>
<td>Tracked_Particle *p</td>
<td>liquid fraction (wet combusting particles only)</td>
</tr>
</tbody>
</table>

#### Table 5.6.3: Macros for Particles at Injection into Domain \( (\text{dpm.h}) \)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_INIT_POS(p)[i]</td>
<td>Tracked_Particle *p int i</td>
<td>position ( i=0,1,2 )</td>
</tr>
<tr>
<td>P_INIT_VEL(p)[i]</td>
<td>Tracked_Particle *p int i</td>
<td>velocity ( i=0,1,2 )</td>
</tr>
<tr>
<td>P_INIT_DIAM(p)</td>
<td>Tracked_Particle *p</td>
<td>diameter</td>
</tr>
<tr>
<td>P_INIT_TEMP(p)</td>
<td>Tracked_Particle *p</td>
<td>temperature</td>
</tr>
<tr>
<td>P_INIT_RH0(p)</td>
<td>Tracked_Particle *p</td>
<td>density</td>
</tr>
<tr>
<td>P_INIT_MASS(p)</td>
<td>Tracked_Particle *p</td>
<td>mass</td>
</tr>
<tr>
<td>P_INIT_LF(p)</td>
<td>Tracked_Particle *p</td>
<td>liquid fraction (wet combusting particles only)</td>
</tr>
</tbody>
</table>
Table 5.6.4: Macros for Particle Cell Index and Thread Pointer (dpm.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_CELL(p)</td>
<td>Tracked_Particle *p</td>
<td>cell index of the cell that the particle is currently in</td>
</tr>
<tr>
<td>P_CELL_THREAD(p)</td>
<td>Tracked_Particle *p</td>
<td>pointer to the thread of the cell that the particle is currently in</td>
</tr>
</tbody>
</table>

Table 5.6.5: Macros for Particle Species, Laws, and User Scalars (dpm.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_EVAP_SPECIES_INDEX(p)</td>
<td>Tracked_Particle *p</td>
<td>evaporating species index in mixture</td>
</tr>
<tr>
<td>P_DEVOL_SPECIES_INDEX(p)</td>
<td>Tracked_Particle *p</td>
<td>devolatilizing species index in mixture</td>
</tr>
<tr>
<td>P_OXID_SPECIES_INDEX(p)</td>
<td>Tracked_Particle *p</td>
<td>oxidizing species index in mixture</td>
</tr>
<tr>
<td>P_PROD_SPECIES_INDEX(p)</td>
<td>Tracked_Particle *p</td>
<td>combustion products species index in mixture</td>
</tr>
<tr>
<td>P_CURRENT_LAW(p)</td>
<td>Tracked_Particle *p</td>
<td>current particle law index</td>
</tr>
<tr>
<td>P_NEXT_LAW(p)</td>
<td>Tracked_Particle *p</td>
<td>next particle law index</td>
</tr>
<tr>
<td>P_USER_REAL(p,i)</td>
<td>Tracked_Particle *p</td>
<td>storage array for user-defined values (indexed by i)</td>
</tr>
</tbody>
</table>
Table 5.6.6: Macros for Particle Material Properties (dpm.h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_MATERIAL(p)</td>
<td>Tracked_Particle *p</td>
<td>material pointer</td>
</tr>
<tr>
<td>DPM_SWELLING_COEFF(p)</td>
<td>Tracked_Particle *p</td>
<td>swelling coefficient for devolatilization</td>
</tr>
<tr>
<td>DPM_LATENT_HEAT(p)</td>
<td>Tracked_Particle *p</td>
<td>latent heat</td>
</tr>
<tr>
<td>DPM_HEAT_OF_PYROLYSIS(p)</td>
<td>Tracked_Particle *p</td>
<td>heat of pyrolysis</td>
</tr>
<tr>
<td>DPM_HEAT_OF_REACTION(p)</td>
<td>Tracked_Particle *p</td>
<td>heat of reaction</td>
</tr>
<tr>
<td>DPM_VOLATILE_FRACTION(p)</td>
<td>Tracked_Particle *p</td>
<td>volatile fraction</td>
</tr>
<tr>
<td>DPM_CHAR_FRACTION(p)</td>
<td>Tracked_Particle *p</td>
<td>char fraction</td>
</tr>
<tr>
<td>DPM_EMISSIVITY(p,m)</td>
<td>Tracked_Particle *p, Material *m</td>
<td>emissivity for radiation model scattering factor</td>
</tr>
<tr>
<td>DPM_SCATT_FACTOR(p,m)</td>
<td>Tracked_Particle *p, Material *m</td>
<td>for radiation model evaporation boiling temperature</td>
</tr>
<tr>
<td>DPM_EVAPORATION_TEMPERATURE(p,m)</td>
<td>Tracked_Particle *p, Material *m</td>
<td>temperature</td>
</tr>
<tr>
<td>DPM_BOILING_TEMPERATURE(p,m)</td>
<td>Tracked_Particle *p, Material *m</td>
<td>specific heat at temperature t</td>
</tr>
<tr>
<td>DPM_SPECIFIC_HEAT(p,t)</td>
<td>Tracked_Particle *p, particle temperature t</td>
<td>specific heat of the material used for the liquid associated with the particle</td>
</tr>
<tr>
<td>DPM LIQUID SPECIFIC HEAT(p,t)</td>
<td>Tracked_Particle *p, particle temperature t</td>
<td>Note: particle temperature is typically determined by P.T(p)</td>
</tr>
</tbody>
</table>

Note: particle temperature is typically determined by P.T(p)
5.7 NO\textsubscript{x} Variables

The macros listed in Table 5.7.1 are useful for defining NO\textsubscript{x} production and reduction rates using DEFINE\_NO\textsubscript{x}\_RATE. These macros can be used to return real NO\textsubscript{x} variables in SI units. The variables are available in both the segregated and the coupled solvers. The macros are defined in the sg\_nox\_h header file, which is included in udf\_h. The argument NOx is the pointer to the NO\textsubscript{x} structure.

Table 5.7.1: Macros for NO\textsubscript{x} Rate (sg\_nox\_h)

<table>
<thead>
<tr>
<th>Name(Arguments)</th>
<th>Argument Types</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOX_EQN(nox)</td>
<td>NOx *nox</td>
<td>NOx pollutant equation ID</td>
</tr>
<tr>
<td>NOX_FRATE(nox)</td>
<td>NOx *nox</td>
<td>NOx production rate</td>
</tr>
<tr>
<td>NOX_RRATE(nox)</td>
<td>NOx *nox</td>
<td>NOx reduction rate</td>
</tr>
<tr>
<td>ARRH(nox, k)</td>
<td>NOx *nox, Rate_const k[3]</td>
<td>Arrhenius rate coefficient</td>
</tr>
<tr>
<td>MOLECON(nox, i)</td>
<td>NOx *nox, int i</td>
<td>species (i) molar concentration</td>
</tr>
</tbody>
</table>

In the argument Rate\_const k[3],

\(k = A, \beta, E\)

where,

\(A =\) Pre-exponential factor
\(\beta =\) Temperature Exponent
\(E =\) Activation Energy
Chapter 6. Utilities

This chapter provides a list predefined utilities that Fluent Inc. has provided for performing operations on FLUENT variables.

- Section 6.1: Introduction
- Section 6.2: Looping Macros
- Section 6.3: Multiphase-Specific Looping Macros
- Section 6.4: Setting Face Variables (F.PROFILE)
- Section 6.5: Accessing Variables That Are Not Passed as Arguments
- Section 6.6: Vector Macros
- Section 6.7: Time-Dependent Macros
- Section 6.8: User-Defined Scheme Variables
- Section 6.9: Input/Output Macros
- Section 6.10: Additional Macros

6.1 Introduction

Many UDF tasks require repeated operations to be performed on all cells or all faces in a thread. For example, defining a custom profile function involves looping over all the faces in a face thread. For your convenience, Fluent Inc. has provided you with a set of macro utilities that perform repeated operations over cells, faces, nodes, and threads. Cell-looping macros can, for example, loop over cells in a given cell thread allowing access to all of the cells. Face-looping macros, on the other hand, can access all faces in a given face thread. These are some of the utilities presented in this chapter. As another example, suppose your UDF is an adjust function and needs to operate on a variable that is not directly passed as an argument (e.g., a thread pointer). Since DEFINE_ADJUST does not have the thread pointer passed to it from the solver, you will need to use a Fluent-provided utility to access the thread pointer in your UDF.

The utilities presented below perform operations on FLUENT variables, and are generally implemented in the code as macros. Many of the macros can be used in UDFs for single-phase and multiphase models, but there are some that are unique to multiphase.
Recall that when you are writing UDFs for multiphase models, you will need to keep in mind the hierarchy of structures within FLUENT (see Section 1.8.1 for details). The particular domain or thread structure that gets passed into your UDF from the solver depends on the DEFINE macro you are using, as well as the domain the function is hooked to (either through the graphical user interface, or hardwired in the code). It also may depend on the multiphase model that you are using. The domain structures that are passed to DEFINE_INIT and DEFINE_ADJUST functions, however, are independent of the multiphase model. These functions are always passed the domain structure associated with the mixture. DEFINE_ON_DEMAND UDFs are not passed any domain structures.

If your UDF is not explicitly passed the pointer for the thread or domain structure required for your function, then you can use the utility macros provided in this chapter to retrieve it. A number of looping macros are also provided that enable your function to loop over all cells and faces for given threads or domains.

### 6.2 Looping Macros

The following looping macros can be used for UDFs in single-phase or multiphase models in FLUENT. Definitions for these macros are contained in the `mem.h` header file.

**Looping over Cell Threads in a Domain (thread_loop_c)**

You can use `thread_loop_c` when you want to loop over all cell threads in a given domain. It consists of a single statement, followed by the operation(s) to be performed on all cell threads in the domain enclosed within braces `{}` as shown below. Note that `thread_loop_c` is similar in implementation to the `thread_loop_f` macro described below.

```c
Domain *domain;
Thread *c_thread;
thread_loop_c(c_thread, domain) /*loops over all cell threads in domain*/
{
}
```
6.2 Looping Macros

Looping over Face Threads in a Domain (thread_loop_f)

You can use thread_loop_f when you want to loop over all face threads in a given domain. It consists of a single statement, followed by the operation(s) to be performed on all face threads in the domain enclosed within braces {} as shown below. Note that thread_loop_f is similar in implementation to the thread_loop_c macro described above.

```c
Thread *f_thread;
Domain *domain;
thread_loop_f(f_thread, domain)/* loops over all face threads in a domain*/
{
}
```

Looping over Cells in a Cell Thread (begin...end_c_loop)

You can use begin_c_loop and end_c_loop when you want to loop over all cells c in a given cell thread c_thread. It contains a begin and end loop statement, and performs operation(s) on each cell in the cell thread as defined between the braces {}. This loop is usually nested within thread_loop_c when you want to loop over all cells in all cell threads in a domain.

```c
cell_t c;
Thread *c_thread;
begin_c_loop(c, c_thread) /* loops over cells in a cell thread */
{
}
end_c_loop(c, c_thread)
```

Example

```c
/* Loop over cells in a thread to get information stored in cells. */
begin_c_loop(c, c_thread)
{
    /* C_T gets cell temperature. The += will cause all of the cell temperatures to be added together. */
    temp += C_T(c, c_thread);
}
end_c_loop(c, c_thread)
```
Looping over Faces in a Face Thread (begin...end_f_loop)

You can use begin_f_loop and end_f_loop when you want to loop over all faces f in a given face thread f_thread. It contains a begin and end loop statement, and performs operation(s) on each face in the face thread as defined between the braces {}. This loop is usually nested within thread_loop_f when you want to loop over all faces in all face threads in a domain.

```c
face_t f;
Thread *f_thread;
begin_f_loop(f, f_thread) /* loops over faces in a face thread */
{
}
end_f_loop(f, f_thread)
```

Example

```c
/* Loop over faces in a face thread to get the information stored on faces. */
begin_f_loop(f, f_thread)
{
    /* F_T gets face temperature. The += will cause all of the face
     * temperatures to be added together. */

    temp += F_T(f, f_thread);
}
end_f_loop(f, f_thread)
```

Looping over Faces on a Cell (c_face_loop)

The following looping function loops over all faces on a given cell. It consists of a single loop statement, followed by the action to be taken in braces {}.

```c
cell_t c;
Thread *t;
face_t f;
Thread *tf;
int n;
c_face_loop(c, t, n) /* loops over all faces on a cell */
{
    .
    .
    .
    f = C_FACE(c,t,n);
```
6.2 Looping Macros

```c
float tf = C_FACE_THREAD(c,t,n);
```

Here, \( n \) is the local face index number. The local face index number is used in the `C_FACE` macro to obtain the global face number (e.g., \( f = C\_FACE(c,t,n) \)).

Another useful macro that is used in `c_face_loop` is `C_FACE_THREAD`. This macro is used to reference the associated face thread (e.g., \( tf = C\_FACE\_THREAD(c,t,n) \)).

Refer to Section 6.10 for other macros that are associated with `c_face_loop`.

### Looping over Nodes of a Cell (c_node_loop)

The following looping function loops over all nodes on a given cell. It consists of a single loop statement, followed by the action to be taken in braces {}.

```c
cell_t c;
Thread *t;
int n;
Node *node;
c_node_loop(c, t, n)
{
  ...
  ...
  ...
  node = C\_NODE(c,t,n);
  ...
  ...
  }
```

Here, \( n \) is the local node index number. The index number can be used with the `C_NODE` macro to obtain the global node number (e.g., \( node = C\_NODE(c,t,n) \)).
6.3 Multiphase-Specific Looping Macros

The following set of macros apply to multiphase model UDFs. Refer to Section 1.8.1 and, in particular, Figure 1.8.1 for a discussion on hierarchy of structures within FLUENT.

Looping over Phase Domains in a Mixture (sub_domain_loop)

The sub_domain_loop macro loops over all phase domains (subdomains) within the mixture domain. The macro steps through and provides each phase domain pointer defined in the mixture domain as well as the corresponding phase_domain_index. As discussed in Section 1.8.1, the domain pointer is needed, in part, to gain access to data within each phase. Note that sub_domain_loop is similar in implementation to the sub_thread_loop macro described below.

```c
int phase_domain_index; /* index of subdomain pointers */
Domain *mixture_domain;
Domain *subdomain;
sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
```

The variable arguments to sub_domain_loop are subdomain, mixture_domain, and phase_domain_index. subdomain is a pointer to the phase-level domain, and mixture_domain is a pointer to the mixture-level domain. The mixture_domain is automatically passed to your UDF by the FLUENT solver when you use a DEFINE macro that contains a domain variable argument (e.g., DEFINE_ADJUST) and your UDF is hooked to the mixture. If the mixture_domain is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (e.g., Get_Domain(1) before calling sub_domain_loop (see Section 6.5). phase_domain_index is an index of subdomain pointers. phase_domain_index is 0 for the primary phase, and is incremented by one for each secondary phase in the mixture. Note that subdomain and phase_domain_index are set within the sub_domain_loop macro.

Example

The following interpreted UDF patches an initial volume fraction for a particular phase in a solution. It is executed once at the beginning of the solution process. The function sets up a spherical volume centered at 0.5, 0.5, 0.5 with a radius of 0.25. A secondary-phase volume fraction of 1 is then patched to the cells within the spherical volume, while the volume fraction for the secondary phase in all other cells is set to 0.
6.3 Multiphase-Specific Looping Macros

UDF for initializing phase volume fraction

#include "udf.h"

/* domain pointer that is passed by INIT function is mixture domain */
DEFINE_INIT(my_init_function, mixture_domain)
{
    int phase_domain_index;
    cell_t cell;
    Thread *cell_thread;
    Domain *subdomain;
    real xc[ND_ND];

    /* loop over all subdomains (phases) in the superdomain (mixture) */
    sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
    {
        /* loop if secondary phase */
        if (DOMAIN_ID(subdomain) == 2)

            /* loop over all cell threads in the secondary phase domain */
            thread_loop_c (cell_thread, subdomain)
            {
                /* loop over all cells in secondary phase cell threads */
                begin_c_loop_all (cell, cell_thread)
                {
                    C_CENTROID(xc, cell, cell_thread);
                    if (sqrt(ND_SUM(pow(xc[0] - 0.5, 2.),
                        pow(xc[1] - 0.5, 2.),
                        pow(xc[2] - 0.5, 2.))) < 0.25)

                        /* set volume fraction to 1 for centroid */
                        C_VOF(cell, cell_thread) = 1.;
                    else
                        /* otherwise initialize to zero */
                        C_VOF(cell, cell_thread) = 0.;
                }
                end_c_loop_all (cell, cell_thread)
            }
        }
    }
}
Utilities

Looping over Phase Threads in a Mixture (sub_thread_loop)

The sub_thread_loop macro loops over all phase-level threads (subthreads) associated with a mixture-level thread. The macro steps through and returns the pointer to each subthread as well as the corresponding phase_domain_index. As discussed in Section 1.8.1, if the subthread pointer is associated with an inlet zone, then the macro will provide the pointers to the face threads associated with the inlet for each of the phases.

```c
int phase_domain_index;
Thread *subthread;
Thread *mixture_thread;
sub_thread_loop(subthread, mixture_thread, phase_domain_index)
```

The variable arguments to sub_thread_loop are subthread, mixture_thread, and phase_domain_index. subthread is a pointer to the phase thread, and mixture_thread is a pointer to the mixture-level thread. The mixture_thread is automatically passed to your UDF by the FLUENT solver when you use a DEFINE macro that contains a thread variable argument (e.g., DEFINE_PROFILE) and your UDF is hooked to the mixture. If the mixture_thread is not explicitly passed to your UDF, you will need to use a utility macro to retrieve it before calling sub_thread_loop. phase_domain_index is an index of subdomain pointers that can be retrieved using the PHASE_DOMAIN_INDEX macro. (See Section 6.5 for details.) The index begins at 0 for the primary phase, and is incremented by one for each secondary phase in the mixture. Note that subthread and phase_domain_index are initialized within the sub_thread_loop macro definition.

Looping over Phase Cell Threads in a Mixture (mp_thread_loop_c)

The mp_thread_loop_c macro loops through all cell threads (at the mixture level) within the mixture domain and provides the pointers of the phase-level (cell) threads associated with each mixture-level thread. This is nearly identical to the thread_loop_c macro (Section 6.2) when applied to the mixture domain. The difference is that, in addition to stepping through each cell thread, the macro also returns a pointer array (pt) that identifies the corresponding phase-level threads. The pointer to the cell thread for the i\textsuperscript{th} phase is pt[i], where i is the phase_domain_index. pt[i] can be used as an argument to macros requiring the phase-level thread pointer. phase_domain_index can be retrieved using the PHASE_DOMAIN_INDEX macro. (See Section 6.5 for details.)

```c
Thread **pt;
Thread *cell_threads;
Domain *mixture_domain;
mp_thread_loop_c(cell_threads, mixture_domain, pt)
```
6.3 Multiphase-Specific Looping Macros

The variable arguments to \texttt{mp\_thread\_loop\_c} are \texttt{cell\_threads}, \texttt{mixture\_domain}, and \texttt{pt}. \texttt{cell\_threads} is a pointer to the cell threads, and \texttt{mixture\_domain} is a pointer to the mixture-level domain. \texttt{pt} is an array pointer whose elements contain pointers to phase-level threads.

\texttt{mixture\_domain} is automatically passed to your UDF by the FLUENT solver when you use a \texttt{DEFINE} macro that contains a domain variable argument (e.g., \texttt{DEFINE\_ADJUST}) and your UDF is hooked to the mixture. If \texttt{mixture\_domain} is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (e.g., \texttt{Get\_Domain(1)}, described in Section 6.5). Note that the values for \texttt{pt} and \texttt{cell\_threads} are set within the looping function.

\texttt{mp\_thread\_loop\_c} is typically used along with \texttt{begin\_c\_loop}. \texttt{begin\_c\_loop} loops over cells in a cell thread. When \texttt{begin\_c\_loop} is nested within \texttt{mp\_thread\_loop\_c}, you can loop over all cells in all phase cell threads within a mixture.

Looping over Phase Face Threads in a Mixture (\texttt{mp\_thread\_loop\_f})

The \texttt{mp\_thread\_loop\_f} macro loops through all face threads (at the mixture level) within the mixture domain and provides the pointers of the phase-level (face) threads associated with each mixture-level thread. This is nearly identical to the \texttt{thread\_loop\_f} macro when applied to the mixture domain. The difference is that, in addition to stepping through each face thread, the macro also returns a pointer array (\texttt{pt}) that identifies the corresponding phase-level threads. The pointer to the face thread for the \texttt{i}th phase is \texttt{pt[i]}, where \texttt{i} is the \texttt{phase\_domain\_index}. \texttt{pt[i]} can be used as an argument to macros requiring the phase-level thread pointer. The \texttt{phase\_domain\_index} can be retrieved using the \texttt{PHASE\_DOMAIN\_INDEX} macro. (See Section 6.5 for details.)

\begin{verbatim}
Thread **pt;
Thread *face_threads;
Domain *mixture_domain;
mp_thread_loop_f(face_threads, mixture_domain, pt)
\end{verbatim}

The variable arguments to \texttt{mp\_thread\_loop\_f} are \texttt{face\_threads}, \texttt{mixture\_domain}, and \texttt{pt}. \texttt{face\_threads} is a pointer to the face threads, and \texttt{mixture\_domain} is a pointer to the mixture-level domain. \texttt{pt} is an array pointer whose elements contain pointers to phase-level threads.

\texttt{mixture\_domain} is automatically passed to your UDF by the FLUENT solver if you are using a \texttt{DEFINE} macro that contains a domain variable argument (e.g., \texttt{DEFINE\_ADJUST}) and your UDF is hooked to the mixture. If \texttt{mixture\_domain} is not explicitly passed to your UDF, you may use another utility macro to retrieve it (e.g., \texttt{Get\_Domain(1)}, described in Section 6.5). Note that the values for \texttt{pt} and \texttt{face\_threads} are set within the looping function.
mp_thread_loop_f is typically used along with begin_f_loop. begin_f_loop loops over faces in a face thread. When begin_f_loop is nested within mp_thread_loop_f, you can loop over all faces in all phase face threads within a mixture.

### 6.4 Setting Face Variables Using F_PROFILE

F_PROFILE can be used to store a boundary condition in memory for a given face and thread, and is typically nested within a face loop. See mem.h for the complete macro definition for F_PROFILE.

F_PROFILE is typically used along with DEFINE_PROFILE; the index i that is an argument to DEFINE_PROFILE is the same variable in F_PROFILE.

**Macro:**

\[ \text{F\_PROFILE}(f, t, i) \]

**Argument types:**

- face_t f
- Thread *t
- int i

**Function returns:**

void

The arguments of F_PROFILE are f, the index of the face face_t; t, a pointer to the face’s thread t; and an integer i. These variables are passed by the FLUENT solver to your UDF.

i is an identifier for the variable that is being set at a particular boundary. For example, an inlet boundary may have a total pressure and a total temperature associated with it (both of which can be described by user-defined functions). One of the inlet boundary variables will be identified in FLUENT by the integer 0, and the other by the integer 1. The values of these integers are set by the solver when you define your boundary conditions using the Boundary Conditions panel in a FLUENT session.

**Sample**

begin_f_loop(f, t)
{
    F_CENTROID(x,f,t);
    y = x[1];
    FPROFILE(f, t, i) = 1.1e5 - y*y/(.0745*.0745)*0.1e5;
}
end_f_loop(f, t)
6.5 Accessing Variables That Are Not Passed as Arguments

For most standard UDFs written for single-phase or multiphase models (e.g., specifying source terms, properties, or custom profiles), variables that your function needs (e.g., domain or thread pointers) are passed directly to your UDF as arguments by the solver in the solution process. For example, if your UDF defines a custom profile for a particular boundary zone (using DEFINE_PROFILE) and is hooked to the appropriate phase or mixture in FLUENT in the relevant boundary condition panel, then appropriate phase or mixture variables will be passed to your function by the solver at run-time. However, not all UDFs are directly passed the arguments that may be required for your function. Recall, for example, that DEFINE_ADJUST and DEFINE_INIT UDFs are passed mixture domain variables, whereas the DEFINE_ON_DEMAND UDFs are not passed any arguments. This section presents utilities that you can use when you need to access variables that are not directly available to your UDF through a DEFINE function.

Get_Domain

You can use the Get_Domain macro to retrieve a domain pointer when it is not explicitly passed as an argument to your UDF.

Get_Domain(domain_id);

domain_id is an integer whose value is 1 for the mixture domain, and is incremented by one for each phase in a multiphase mixture. Note that Get_Domain(1) replaces the extern Domain *domain expression used in previous releases of FLUENT.

Single-Phase Flows

In the case of single-phase flows, domain_id is 1 and Get_Domain(1) will return the fluid domain pointer.

DEFINE_ON_DEMAND(my_udf)
{
    Domain *domain; /* domain is declared as a variable */
    domain = Get_Domain(1); /* returns fluid domain pointer */
    ...
}
Utilities

Multiphase Flows

In the case of multiphase flows, the value returned by Get_Domain is either the mixture-level, a phase-level, or an interaction phase-level domain pointer. The value of domain_id is always 1 for the mixture domain. You can obtain the domain_id using the FLUENT graphical user interface much in the same way that you can determine the zone ID from the Boundary Conditions panel. Simply go to the Phases panel in FLUENT and select the desired phase. The domain_id will then be displayed. You will need to hard code this integer ID as an argument to the macro as shown below.

DEFINE_ON_DEMAND(my_udf)
{
    Domain *mixture_domain;
    mixture_domain = Get_Domain(1);  /* returns mixture domain pointer */
    /* and assigns to variable */

    Domain *subdomain;
    subdomain = Get_Domain(2);  /* returns phase with ID=2 domain pointer*/
    /* and assigns to variable */

    ...
}

Example

Below is a UDF named get_coords that prints the thread face centroids for two specified thread IDs. The function implements the Get_Domain utility for a single-phase application. In this example, the function Print_Thread_Face_Centroids uses the Lookup_Thread function to determine the pointer to a thread, and then writes the face centroids of all the faces in a specified thread to a file. The Get_Domain(1) function call returns the pointer to the domain (or mixture domain, in the case of a multiphase application). This argument is not passed to DEFINE_ON_DEMAND.

/* Example of UDF for single phase that uses Get_Domain utility */
#include "udf.h"

FILE *fout;

void Print_Thread_Face_Centroids(Domain *domain, int id)
{
    real FC[2];
    face_t f;
Thread *t = Lookup_Thread(domain, id);

fprintf(fout,"thread id %d\n", id);
begin_f_loop(f,t)
{  
  F_CENTROID(FC,f,t);
  fprintf(fout, "f%d %g %g %g\n", f, FC[0], FC[1], FC[2]);
}  
end_f_loop(f,t)
fprintf(fout, "\n");
}

DEFINE_ON_DEMAND(get_coords)
{
  Domain *domain;
  domain = Get_Domain(1);
  fout = fopen("faces.out", "w");
  Print_Thread_Face_Centroids(domain, 2);
  Print_Thread_Face_Centroids(domain, 4);
  fclose(fout);
}

Phase Domain Pointer Using the Phase Domain Index
(DOMAIN_SUB_DOMAIN)

There are two ways you can get access to a specific phase (or subdomain) pointer within
the mixture domain. You can use either the DOMAIN_SUB_DOMAIN macro (described below)
or Get_Domain, which is described below.

DOMAIN_SUB_DOMAIN has two arguments: mixture_domain and phase_domain_index. The
function returns the phase pointer subdomain for the given phase_domain_index. Note
that DOMAIN_SUB_DOMAIN is similar in implementation to the THREAD_SUB_THREAD macro
described in Section 6.5.

int phase_domain_index = 0;  /* primary phase index is 0 */
Domain *mixture_domain;
Domain *subdomain = DOMAIN_SUB_DOMAIN(mixture_domain,phase_domain_index);
mixture_domain is a pointer to the mixture-level domain. It is automatically passed to
your UDF by the FLUENT solver when you use a DEFINE macro that contains a domain
variable argument (e.g., DEFINE_ADJUST) and your UDF is hooked to the mixture. Oth-
erwise, if the mixture_domain is not explicitly passed to your UDF, you will need to use
another utility macro to retrieve it (e.g., Get_Domain(1)) before calling sub_domain_loop.
Utilities

phase_domain_index is an index of subdomain pointers. It is an integer that starts with 0 for the primary phase and is incremented by one for each secondary phase. phase_domain_index is automatically passed to your UDF by the FLUENT solver when you use a DEFINE macro that contains a phase domain index argument (DEFINE.Exchange_Property, DEFINE_Vector.Exchange_Property) and your UDF is hooked to a specific interaction phase. Otherwise, you will need to hard code the integer value of phase_domain_index to the DOMAIN_SUB_DOMAIN macro. If your multiphase model has only two phases defined, then phase_domain_index is 0 for the primary phase, and 1 for the secondary phase. However, if you have more than one secondary phase defined for your multiphase model, you will need to use the PHASE_DOMAIN_INDEX utility to retrieve the corresponding phase_domain_index for the given domain. See Section 6.5 for details.

Phase-Level Thread Pointer Using the Phase Domain Index
(THREAD_SUB_THREAD)

The THREAD_SUB_THREAD macro can be used to retrieve the phase-level thread (sub-thread) pointer, given the phase domain index. THREAD_SUB_THREAD has two arguments: mixture_thread and phase_domain_index. The function returns the phase-level thread pointer for the given phase_domain_index. Note that THREAD_SUB_THREAD is similar in implementation to the DOMAIN_SUB_DOMAIN macro described in Section 6.5.

```c
int phase_domain_index = 0; /* primary phase index is 0 */
Thread *mixture_thread; /* mixture-level thread pointer */
Thread *subthread = THREAD_SUB_THREAD(mixture_thread, phase_domain_index);
```

mixture_thread is a pointer to a mixture-level thread. It is automatically passed to your UDF by the FLUENT solver when you use a DEFINE macro that contains a variable thread argument (e.g., DEFINE_PROFILE), and the function is hooked to the mixture. Otherwise, if the mixture thread pointer is not explicitly passed to your UDF, then you will need to use the Lookup_Thread utility macro to retrieve it (see Section 6.5).

phase_domain_index is an index of subdomain pointers. It is an integer that starts with 0 for the primary phase and is incremented by one for each secondary phase. phase_domain_index is automatically passed to your UDF by the FLUENT solver when you use a DEFINE macro that contains a phase domain index argument (DEFINE.Exchange_Property, DEFINE_Vector.Exchange_Property) and your UDF is hooked to a specific interaction phase. (See Section 4.4.2 for an example UDF.) Otherwise, you will need to hard code the integer value of phase_domain_index to the THREAD_SUB_THREAD macro. If your multiphase model has only two phases defined, then phase_domain_index is 0 for the primary phase, and 1 for the secondary phase. However, if you have more than one secondary phase defined for your multiphase model, you will need to use the PHASE_DOMAIN_INDEX utility to retrieve the corresponding phase_domain_index for the given domain. See Section 6.5 for details.
6.5 Accessing Variables That Are Not Passed as Arguments

Phase Thread Pointer Array Using Mixture-Level Thread
(THREAD_SUB_THREADS)

The THREAD_SUB_THREADS macro can be used to retrieve the pointer array, pt, whose
elements contain pointers to phase-level threads (subthreads). THREADS_SUB_THREADS
has one argument, mixture_thread.

Thread *mixture_thread;
Thread **pt; /* initialize pt */
pt = THREAD_SUB_THREADS(mixture_thread);

mixture_thread is a pointer to a mixture-level thread which can represent a cell thread
or a face thread. It is automatically passed to your UDF by the FLUENT solver when you
use a DEFINE macro that contains a variable thread argument (e.g., DEFINE_PROFILE),
and the function is hooked to the mixture. Otherwise, if the mixture thread pointer is
not explicitly passed to your UDF, then you will need to use another method to retrieve
it. For example you can use the Lookup_Thread utility macro (see Section 6.5).

pt[i], an element in the array, is a pointer to the corresponding phase-level thread for
the i-th phase, where i is the phase_domain_index. You can use pt[i] as an argument
to some cell variable macros when you want to retrieve specific phase information at a
cell. For example, C_R(c,pt[i]) can be used to return the density of the i-th phase fluid
at cell c. The pointer pt[i] can also be retrieved using THREAD_SUB_THREAD, discussed
in Section 6.5, using i as an argument. The phase_domain_index can be retrieved using
the PHASE_DOMAIN_INDEX macro. See Section 6.5 for details.

Mixture Domain Pointer Using a Phase Domain Pointer
(DOMAIN_SUPER_DOMAIN)

You can use DOMAIN_SUPER_DOMAIN when your UDF has access to a particular phase-level
domain (subdomain) pointer, and you want to retrieve the mixture-level domain pointer.
DOMAIN_SUPER_DOMAIN has one argument, subdomain. Note that DOMAIN_SUPER_DOMAIN
is similar in implementation to the THREAD_SUPER_THREAD macro described in Section 6.5.

Domain *subdomain;
Domain *mixture_domain = DOMAIN_SUPER_DOMAIN(subdomain);

subdomain is a pointer to a phase-level domain within the multiphase mixture. It is
automatically passed to your UDF by the FLUENT solver when you use a DEFINE macro
that contains a domain variable argument (e.g., DEFINE_ADJUST), and the function is
hooked to a primary or secondary phase in the mixture. Note that in the current ver-
sion of FLUENT, DOMAIN_SUPER_DOMAIN will return the same pointer as Get_Domain(1).
Therefore, if a subdomain pointer is available in your UDF, it is recommended that the
Utilities

 DOMAIN_SUPER_DOMAIN macro be used instead of the Get_Domain macro to avoid potential incompatibility issues with future releases of FLUENT.

Mixture Thread Pointer Using a Phase Thread Pointer (THREAD_SUPER_THREAD)

You can use the THREAD_SUPER_THREAD macro when your UDF has access to a particular phase-level thread (subthread) pointer, and you want to retrieve the mixture-level thread pointer. THREAD_SUPER_THREAD has one argument, subthread.

Thread *subthread;
Thread *mixture_thread = THREAD_SUPER_THREAD(subthread);

subthread is a pointer to a particular phase-level thread within the multiphase mixture. It is automatically passed to your UDF by the FLUENT solver when you use a DEFINE macro that contains a thread variable argument (e.g., DEFINE_PROFILE, and the function is hooked to a primary or secondary phase in the mixture. Note that THREAD_SUPER_THREAD is similar in implementation to the DOMAIN_SUPER_DOMAIN macro described in Section 6.5.

Thread Pointer Using a Zone ID (Lookup_Thread)

You can use the Lookup_Thread macro when you want to retrieve the thread pointer that corresponds to a given zone ID from the Boundary Conditions panel in FLUENT. For example, suppose that your UDF needs to operate on a particular thread in a domain (instead of looping over all threads), and the DEFINE macro you are using to define your UDF doesn’t have the thread pointer passed to it from the solver (e.g., DEFINE_ADJUST). Your UDF can use Lookup_Thread to get the desired thread pointer. This is a two-step process. First, you will need to get the integer ID of the zone (1, 2, etc.) from the Boundary Conditions panel in FLUENT. Then you will need to hard code the zone_ID as an argument to the Lookup_Thread macro. Lookup_Thread returns the pointer to the thread that is associated with the given zone ID. You can then assign the thread pointer to a thread_name and use it in your UDF.

int zone_ID;
Thread *thread_name = Lookup_Thread(domain,zone_ID);

In the context of multiphase flows, the thread returned by the Lookup_Thread macro will be at the phase level associated with the domain argument.

Example

Below is a UDF that uses the Lookup_Thread macro. In this example, the pointer to a thread for a given zone_ID is retrieved by Lookup_Thread and is assigned to thread.
6.5 Accessing Variables That Are Not Passed as Arguments

The thread pointer is then used in `begin_f_loop` and `F_CENTROID` to loop over all faces in the given thread, get the face centroid value, and print it to a file.

```c
#include "udf.h"

/* domain passed to Adjust function is mixture domain for multiphase*/
DEFINE_ADJUST(print_f_centroids, domain)
{
    real FC[2];
    face_t f;
    int ID = 1;
    /* Zone ID for wall-1 zone from Boundary Conditions panel */
    Thread *thread = Lookup_Thread(domain, ID);
    begin_f_loop(f, thread)
    {
        F_CENTROID(FC, f, thread);
        printf("x-coord = %f y-coord = %f", FC[0], FC[1]);
    }
    end_f_loop(f, thread)
}
```

**domain_id Using a Phase Domain Pointer (DOMAIN_ID)**

You can use `DOMAIN_ID` when you want to access the `domain_id` that corresponds to a given phase-level domain pointer. `DOMAIN_ID` has one argument, `subdomain`, which is the pointer to a phase-level domain. The default `domain_id` value for the top-level domain (mixture) is 1. That is, if the domain pointer that is passed to `DOMAIN_ID` is the mixture-level domain pointer, then the function will return a value of 1. Note that the `domain_id` that is returned by the macro is the same integer ID that is displayed in the graphical user interface when you select the desired phase in the Phases panel in FLUENT.

```c
Domain *subdomain;
int domain_id = DOMAIN_ID(subdomain);
```
Phase Domain Index Using a Phase Domain Pointer
(PHASE_DOMAIN_INDEX)

The PHASE_DOMAIN_INDEX macro retrieves the phase_domain_index for a given phase-level domain (subdomain) pointer. PHASE_DOMAIN_INDEX has one argument, subdomain, which is the pointer to a phase-level domain. phase_domain_index is an index of subdomain pointers. It is an integer that starts with 0 for the primary phase and is incremented by one for each secondary phase.

```c
Domain *subdomain;
int phase_domain_index = PHASE_DOMAIN_INDEX(subdomain);
```

6.6 Vector Macros

Fluent Inc. has provided you with some utilities that you can use in your UDFs to access or manipulate vector quantities in FLUENT. These utilities are implemented as macros in the code. For example, you can use the real function NV_MAG(V) to compute the magnitude of vector V. Alternatively, you can use the real function NV_MAG2(V) to obtain the square of the magnitude of vector V. Below is a list of vector utilities that you can use in your UDF.

There is a naming convention for vector utility macros. V denotes a vector, S denotes a scalar, and D denotes a sequence of three vector components of which the third is always ignored for a two-dimensional calculation. The standard order of operations convention of parentheses, exponents, multiplication, division, addition, and subtraction (PEMDAS) is not followed in vector functions. Instead, the underscore (_) sign is used to group operands into pairs, so that operations are performed on the elements of pairs before they are performed on groups.

! Note that all of the vector utilities in this section have been designed to work correctly in 2D and 3D. Consequently, you don’t need to do any testing to determine this in your UDF.

NV_MAG

The utility NV_MAG computes the magnitude of a vector. This is taken as the square root of the sum of the squares of the vector components.

```c
NV_MAG(x)
```

```c
2D:  sqrt(x[0]*x[0] + x[1]*x[1]);
3D:  sqrt(x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
```
6.6 Vector Macros

**NV_MAG2**

The utility **NV_MAG2** computes the sum of squares of vector components.

\[ NV_MAG2(x) \]

- **2D:** \( x[0]^2 + x[1]^2 \)
- **3D:** \( x[0]^2 + x[1]^2 + x[2]^2 \)

**ND ND**

The constant **ND ND** is defined as 2 for **RP_2D** (FLUENT 2D) and **RP_3D** (FLUENT 3D). It can be used when you want to build a \( 2 \times 2 \) matrix in 2D and a \( 3 \times 3 \) matrix in 3D. When you use **ND ND**, your UDF will work for both 2D and 3D cases, without requiring any modifications.

\[ \text{real A[ND ND][ND ND]} \]

\[
\text{for (i=0; i<ND ND; ++i)} \\
\text{for (j=0; j<ND ND; ++j)} \\
\text{A[i][j] = f(i, j);} \\
\]

**ND_SUM**

The utility **ND_SUM** computes the sum of **ND ND** arguments.

\[ ND_SUM(x, y, z) \]

- **2D:** \( x + y \)
- **3D:** \( x + y + z \)
ND_SET

The utility ND_SET generates ND ND assignment statements.

\[
\text{ND_SET}(u, v, w, C_U(c, t), C_V(c, t), C_W(c, t))
\]

\[
u = C_U(c, t);
\]
\[
v = C_V(c, t);
\]
\[
\text{if 3D:}
\]
\[
w = C_W(c, t);
\]

NV_V

The utility NV_V performs an operation on two vectors.

\[
\text{NV_V}(a, =, x);
\]
\[
a[0] = x[0]; a[1] = x[1]; \text{ etc.}
\]

Note that if you use + = instead of = in the above equation, then you get

\[
a[0]+=x[0]; \text{ etc.}
\]

NV_VV

The utility NV_VV performs operations on vector elements. The operation that is performed on the elements depends upon what symbol (-,/,*) is used as an argument in place of the + signs in the following macro call.

\[
\text{NV_VV}(a, =, x, +, y)
\]
\[
2D: \ a[0] = x[0] + y[0], a[1] = x[1] + y[1];
\]

NV_V_VS

The utility NV_V_VS adds a vector to another vector which is multiplied by a scalar.

\[
\text{NV_V_VS}(a, =, x, +, y, *, 0.5);
\]
\[
2D: \ a[0] = x[0] + (y[0]*0.5), a[1] = x[1] +(y[1]*0.5);
\]

Note that the + sign can be replaced by -, /, or *, and the * sign can be replaced by /.
NV_VS_VS

The utility NV_VS_VS adds a vector to another vector which are each multiplied by a scalar.

```c
NV_VS_VS(a, =, x, *, 2.0, +, y, *, 0.5);
```

2D: \( a[0] = (x[0] \times 2.0) + (y[0] \times 0.5) \), \( a[1] = (x[1] \times 2.0) + (y[1] \times 0.5) \);

Note that the + sign can be used in place of -, *, or /, and the * sign can be replaced by /.

ND_DOT

The following utilities compute the dot product of two sets of vector components.

```c
ND_DOT(x, y, z, u, v, w)
```

2D: \( (x \times u + y \times v) \);
3D: \( (x \times u + y \times v + z \times w) \);

```c
NV_DOT(x, u)
```

2D: \( (x[0] \times u[0] + x[1] \times u[1]) \);
3D: \( (x[0] \times u[0] + x[1] \times u[1] + x[2] \times u[2]) \);

```c
NVD_DOT(x, u, v, w)
```

2D: \( (x[0] \times u + x[1] \times v) \);
3D: \( (x[0] \times u + x[1] \times v + x[2] \times w) \);
6.7 Time-Dependent Macros

You can access time-dependent variables in your UDF in two different ways: direct access using a solver macro, or indirect access using an RP variable macro. Below is a list of solver macros that you can use to access time-dependent variables in FLUENT (Table 6.7.1). An example of a UDF that uses a solver macro to access a time-dependent variable is provided below.

Table 6.7.1: Solver Macros for Time-Dependent Variables

<table>
<thead>
<tr>
<th>Macro Name</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_TIME</td>
<td>real current flow time (in seconds)</td>
</tr>
<tr>
<td>CURRENT_TIMESTEP</td>
<td>real current physical time step size (in seconds)</td>
</tr>
<tr>
<td>PREVIOUS_TIME</td>
<td>real previous flow time (in seconds)</td>
</tr>
<tr>
<td>PREVIOUS_2_TIME</td>
<td>real flow time two steps back in time (in seconds)</td>
</tr>
<tr>
<td>PREVIOUS_TIMESTEP</td>
<td>real previous physical time step size (in seconds)</td>
</tr>
<tr>
<td>N_TIME</td>
<td>integer number of time steps</td>
</tr>
<tr>
<td>N_ITER</td>
<td>integer number of iterations</td>
</tr>
</tbody>
</table>

There are some time-dependent variables, such as current physical flow time, that can be accessed directly using a solver macro (CURRENT_TIME), or indirectly by means of the RP variable macro RP_Get_Real("flow-time").

Solver Macro Usage

```c
real current_time;
current_time = CURRENT_TIME;
```

“Equivalent” RP Macro Usage

```c
real current_time;
current_time = RP_Get_Real("flow-time");
```

Table 6.7.2 shows the correspondence between solver and RP macros that access the same time-dependent variables.
Table 6.7.2: Solver and RP Macros that Access the Same Time-Dependent Variable

<table>
<thead>
<tr>
<th>Solver Macro</th>
<th>“Equivalent” RP Variable Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_TIME</td>
<td>RP_Get_Real(&quot;flow-time&quot;)</td>
</tr>
<tr>
<td>CURRENT_TIMESTEP</td>
<td>RP_Get_Real(&quot;physical-time-step&quot;)</td>
</tr>
<tr>
<td>N_TIME</td>
<td>RP_Get_Integer(&quot;time-step&quot;)</td>
</tr>
</tbody>
</table>

Example

The integer time step count (accessed using N_TIME) is useful in DEFINE_ADJUST functions for detecting whether the current iteration is the first in the time step. This is shown in the following example.

```c
/******************************************************************************
   Example UDF that uses N_TIME
*******************************************************************************/
static int last_ts = -1;  /* Global variable. Time step is never <0 */
DEFINE_ADJUST(first_iter_only, domain)
{
    int curr_ts;
    curr_ts = N_TIME;
    if (last_ts != curr_ts)
    {
        last_ts = curr_ts;

        /* things to be done only on first iteration of each time step
         * can be put here */
    }
}
```
6.8 User-Defined Scheme Variables

The text interface of FLUENT executes a Scheme interpreter which allows you to define your own variables that can be stored in FLUENT and accessed via your UDF. This capability can be very useful, for example, if you want to alter certain parameters in your case, and you do not want to recompile your UDF each time. For example, suppose you want to apply a UDF to multiple zones in a grid. You can do this manually by accessing a particular Zone ID in the graphical user interface, hardcoding the integer ID in your UDF, and then recompiling the UDF. This can be a tedious process if you want to apply your UDF to a number of zones. By defining your own scheme variable, if you want to alter the variable later, then you can do it from the text interface using a Scheme command.

Macros that are used to specify and access user-specified Scheme variables are identified by the prefix `rp`, (e.g., `rp-var-define`), and are executed in the text interface. Macros that are used to access user-defined Scheme variables in the solver are identified by the prefix `RP` (e.g., `RP_Get_Real`), and are executed in UDFs.

Defining Your Own Scheme Variable

Before you define a scheme variable (e.g., `pres_av/thread-id`), it is often good to check that the variable is not already defined. You can do this by typing the following command in the text window:

```
(if (not (rp-var-object 'pres_av/thread-id))
     (rp-var-define 'pres_av/thread-id 2 'integer #f))
```

This command first checks that the variable `pres_av/thread-id` is not already defined, and then sets it up as an integer with an initial value of 2.

Note that the string `/` is allowed in Scheme variable names (as in `pres_av/thread-id`), and is a useful way to organize variables so that they do not interfere with each other.

Accessing a Scheme Variable in the Text Interface

Once you define a Scheme variable in the text interface, you can access the variable. For example, if you want to check the current value of the variable (e.g., `pres_av/thread-id`) on the Scheme side, you can type the following command in the text window:

```
(rpgetvar 'pres_av/thread-id)
```
6.8 User-Defined Scheme Variables

Changing a Scheme Variable to Another Value in the Text Interface

Alternatively, if you want to change the value of the variable you have defined (pres_av/thread-id) to say, 7, then you will need to use `rpsetvar` and issue the following command in the text window:

```
(rpsetvar 'pres_av/thread-id 7)
```

Accessing a Scheme Variable in Your UDF

Once your new variable is defined on the Scheme side, you will need to bring it over to the solver side to be able to use it in your UDF. Fluent macros, referred to as ‘RP’ macros are used to access Scheme variables in C, and are listed below.

- `RP_Get_Real("variable-name")` Returns the double value of `variable-name`
- `RP_Get_Integer("variable-name")` Returns the integer value of `variable-name`
- `RP_Get_String("variable-name")` Returns the char* value of `variable-name`
- `RP_Get_Boolean("variable-name")` Returns the Boolean value of `variable-name`

For example, to access the user-defined Scheme variable `pres_av/thread-id` in your C function, you will use `RP_Get_Integer`. You can then assign the variable returned to a local variable you have declared in your UDF (e.g., `surface_thread_id`) as demonstrated below:

```
surface_thread_id = RP_Get_Integer("pres_av/thread-id");
```
6.9 Input/Output Macros

Fluent Inc. has provided some utilities that you can use to perform input/output (I/O) tasks. These are listed below and are described in the following sections:

Message(format, ...) prints a message to the console window
Error(format, ...) prints an error message to the console window

Message

The Message macro is a utility that displays data to the console in a format that you specify.

int Message(char *format, ...);

The first argument in the Message macro is the format string. It specifies how the remaining arguments are to be displayed in the console window. The format string is defined within quotes. The value of the replacement variables that follow the format string will be substituted in the display for all instances of %type. The % character is used to designate the character type. Some common format characters are: %d for integers, %f for floating point numbers, and %e for floating point numbers in exponential format (with e before the exponent). Consult a C programming language manual for more details. The format string for Message is similar to printf, the standard C I/O function (see Section 2.13.3 for details).

In the example below, the text Volume integral of turbulent dissipation: will be displayed in the console window, and the value of the replacement variable, sum_diss, will be substituted in the message for all instances of %g.

Example:

Message("Volume integral of turbulent dissipation: %g\n", sum_diss);
   /* g represents floating point number in f or e format */
   /* \n denotes a new line */

! It is recommended that you use Message instead of printf in compiled UDFs (UNIX only).
Error

You can use Error when you want to stop execution of a UDF and print an error message to the console window.

Example:

```c
if (table_file == NULL)
    Error("error reading file");
```

Error is not supported by the interpreter and can only be used in compiled UDFs.

6.10 Additional Macros

N_UDS()

You can use N_UDS() to access the number of user-defined scalar transport equations that have been specified in FLUENT. The macro takes no arguments and returns the integer number of equations. It is defined in models.h.

N_UDM()

You can use N_UDM() to access the number of user-defined memory locations that have been used in FLUENT. The macro takes no arguments, and returns the integer number of memory locations used. It is defined in models.h.

Data_Valid_P()

You can check that the cell values of the variables that appear in your function are accessible before you compute the function by using the Data_Valid_P macro.

```c
boolean Data_Valid_P()
```

Data_Valid_P is defined in the id.h header file, and is included in udf.h. The function returns 1 (true) if the data that is passed as an argument is valid, and 0 (false) if it is not.

Example:

```c
if(!Data_Valid_P()) return;
```
For example, suppose you read a case file and, in the process, load a UDF. If that function performs a calculation using variables that have not yet been initialized, such as the velocity at interior cells, an error will occur. To avoid this kind of error, an if else condition can be added to your code. If (if) the data are available, the function can be computed in the normal way. If not (else), no (or some trivial) calculation can be performed instead. Once the flow field has been initialized, the function can be reinvoked so that the correct calculation can be performed.

**FLUID_THREAD_P()**

```java
boolean FLUID_THREAD_P(t);
```

You can use `FLUID_THREAD_P` to check whether a cell thread is a fluid thread. The macro is passed a cell thread pointer `t`, and returns 1 (or `TRUE`) if the thread that is passed is a fluid thread, and 0 (or `FALSE`) if it is not.

Note that `FLUID_THREAD_P` assumes that the thread is a cell thread.

For example,

```java
FLUID_THREAD_P(t0);
```

returns `TRUE` if the thread pointer `t0` passed as an argument represents a fluid thread.

**NULLP & NNULLP**

You can use the `NULLP` and `NNULLP` functions to check whether storage has been allocated for user-defined scalars. `NULLP` returns `TRUE` if storage is not allocated, and `NNULLP` returns `TRUE` if storage is allocated. Below are some examples of usage.

```java
NULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))
/* NULLP returns TRUE if storage is not allocated for user-defined storage variable */

NNULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))
/* NNULLP returns TRUE if storage is allocated for user-defined storage variable */
```
M_PI

The macro M_PI returns the value of π.

UNIVERSAL_GAS_CONSTANT

The macro UNIVERSAL_GAS_CONSTANT returns the value of the universal gas constant (8314.34), which is expressed in SI units of J/Kmol-K.

SQR(k)

SQR(k) returns the square of the given variable k, or \( k \times k \).
Chapter 7. Interpreting and Compiling Source Files

Once you have written your UDF using any text editor and have saved it with a .c extension in your working directory, you are ready to interpret, or compile and load it. Follow the instructions in Section 7.2 or Section 7.3, depending on whether your UDF will be interpreted or compiled in FLUENT.

- Section 7.1: Introduction
- Section 7.2: Interpreting a Source File Using the Interpreted UDFs Panel
- Section 7.3: Compiling a Source File Using the Compiled UDFs Panel

7.1 Introduction

Source code files containing UDFs can be either interpreted or compiled in FLUENT. In both cases the functions are compiled, but the way in which the source code is compiled, and the code that results from the compilation process, is different for the two methods.

7.1.1 Interpreted UDFs

Interpreted UDFs are interpreted directly from a source file (e.g., udfexample.c) at runtime. The process involves a visit to the Interpreted UDFs panel where you can interpret the UDFs in your source file (e.g., udfexample.c) in a single step. Once a source file is interpreted, you can write the case file and the names and contents of your C function(s) will be stored in the case file. In this way, the function(s) will be automatically interpreted whenever the case file is subsequently read. Once interpreted (either manually through the Interpreted UDFs panel or automatically upon reading a a case file), all of the interpreted UDFs that are contained within the source file will become visible and selectable in graphical user interface panel(s) in FLUENT.

Inside FLUENT, the source code is compiled into an intermediate, architecture-independent machine code using a C preprocessor. This machine code then executes on an internal emulator, or interpreter, when the UDF is invoked. This extra layer of code incurs a performance penalty, but allows an interpreted UDF to be shared effortlessly between different architectures, operating systems, and FLUENT versions. If execution speed does become an issue, an interpreted UDF can always be run in compiled mode without modification.
7.1.2 Compiled UDFs

The compilation process for compiled UDFs consists of two steps: build and load. The process involves a visit to the Compiled UDFs panel where you first Build a shared library from one or more source files, and then Load the shared library (e.g., libudf) into FLUENT. Once a shared library is loaded, it can be written with any case file so that it will be automatically loaded whenever that case file is subsequently read. This saves having to reload the compiled library every time you want to run a simulation. Once loaded (either manually through the Compiled UDFs panel or automatically upon reading a case file), all of the compiled UDFs that are contained within the shared library, will become visible and selectable in graphical user interface panel(s) in FLUENT.

Compiled UDFs are built in the same way that the FLUENT executable itself is built. Internally, a script called Makefile is used to invoke the system C compiler to build an object code library that contains the native machine language translation of your higher-level C source code. This shared library (e.g., libudf) is then loaded into FLUENT (either at runtime or automatically when a case file is read) by a process called “dynamic loading.” The object libraries are specific to the computer architecture being used, as well as to the particular version of the FLUENT executable being run. The libraries must, therefore, be rebuilt any time FLUENT is upgraded, when the computer’s operating system level changes, or when the job is run on a different type of computer.

7.1.3 Location of the udf.h File

UDFs are defined using DEFINE macros (see Chapter 4) and the definitions for DEFINE macros are included in udf.h. Consequently, before you interpret or compile your source file, the udf.h header file will need to be accessible in your path, or saved locally within your working directory.

The location of the udf.h file is:

\[ \text{path/Fluent.Inc/fluent6.x/src/udf.h} \]

where \text{path} is the directory in which you have installed the release directory, Fluent.Inc, and \text{x} is replaced by the appropriate number for the release you have (e.g., 1 for fluent6.1).

! In general, you should not copy udf.h from the installation area. The compiler is designed to look for this file locally (in your current directory) first. If it is not found in your current directory, the compiler will look in the /src directory automatically. In the event that you upgrade your release area, but do not remove an old copy of udf.h from your working directory, you will not be accessing the most recent version of this file.

! You should not, under any circumstances, alter the udf.h file.
7.2 Interpreting a Source File Using the Interpreted UDFs Panel

This section presents the steps for interpreting a source file in FLUENT. Once interpreted, the names of UDFs contained within the source file will appear in drop-down lists in graphical user interface panels in FLUENT.

The general procedure for interpreting a source file is as follows:

1. Make sure that the UDF source file(s) are in the same directory that contains your case and data files.
   - If you are running the parallel version of FLUENT on a network of Windows machines, you must 'share' the working directory that contains your UDF source, case, and data files so that all of the compute nodes in the cluster can see it. To do this:
     - Open the Windows Explorer application, right click on the folder for the working directory (e.g., mywork), select the Sharing... option, and specify a Share Name (e.g., mywork).

2. Start FLUENT from your working directory.

3. Read (or set up) your case file.

4. Open the Interpreted UDFs panel (Figure 7.2.1).

   ![Interpreted UDFs Panel]

   **Figure 7.2.1: The Interpreted UDFs Panel**
5. In the Interpreted UDFs panel, select the UDF source file(s) you want to interpret by either typing the complete path in the Source File Name field or use the browser.

(a) To select your source file(s) using the browser, click Browse... under Source Files in the Interpreted UDFs panel. This will open the Select File panel (Figure 7.2.2).

![Select File Panel](image)

Figure 7.2.2: The Select File Panel

(b) In the Select File panel, highlight the directory path under Directories (e.g., /nfs/homeserver/home/clb/mywork/), and the desired file (e.g., udfexample.c) under Files, and click OK.

If you are running FLUENT on a network of Windows machines, you may need to type the file’s complete path in the Source File Name field, instead of using the browser option. For example, to interpret udfexample.c that is located in a shared working directory named mywork, you would enter the following:

```
\<fileserver>\mywork\udfexample.c
```

in the Source File Name field in the Interpreted UDFs panel, replacing <fileserver> with the name of the computer on which your working directory (mywork) and source file (udfexample.c) are located.
6. In the Interpreted UDFs panel, specify the C preprocessor to be used in the CPP Command Name field. You can keep the default cpp or you can select Use Contributed CPP to use the preprocessor supplied by Fluent Inc.

! If you installed the /contrib component from the “PrePost” CD, then by default, the cpp preprocessor will appear in the panel. For Windows NT users, the standard Windows NT installation of the FLUENT product includes the cpp preprocessor.

For UNIX systems, there are other ANSI C preprocessors that may be available on your system, including gcc -E and cc -E. Check with your computer system administrator for details.

For Windows NT systems, if you are using the Microsoft compiler, then use the command cl -E. Check with your computer system administrator for details.

7. Keep the default Stack Size setting of 10000, unless the number of local variables in your function will cause the stack to overflow. In this case, set the Stack Size to a number that is greater than the number of local variables used.

8. Keep the Display Assembly Listing option on if you want a listing of assembly language code to appear in your console window when the function interprets. This option will be saved in your case file, so that when you read the case in a subsequent FLUENT session, the assembly code will be automatically displayed.

9. Click Interpret to interpret your UDF.

```
inlet_x_velocity:
    .local.pointer thread (r0)
    .local.int nv (r1)
    0 .local.end
    0 save
    1 push.int 0
    .local(pointer x (r4)
    3 begin.data 8 bytes, 0 bytes initialized:
    7 save
    . .
    . .
    . .
    156 pre.inc.int f (r3)
    158 pop.int
    159 b .L3 (22)
    .L2:
    161 restore
    162 restore
    163 ret.v
```
Note that if your compilation is unsuccessful, then FLUENT will report an error and you will need to debug your program. See Section 7.2.1.

10. Close the Interpreted UDFs panel when the interpreter has finished.

11. Write the case file if you want the interpreted function(s) (e.g., inlet_x_velocity) to be saved with the case, and automatically interpreted when the case is subsequently read. If the Display Assembly Listing option was chosen, then the assembly code will appear in the console window.

### 7.2.1 Common Errors Made While Interpreting Your Source File

If there are compilation errors when you interpret your UDF, they will appear in the console window. However, you may not see all the error messages if they scroll off the screen too quickly. For this reason, you may want to turn off the Display Assembly Listing option while debugging your UDF. You can view the compilation history in the 'log' file that is saved in your working directory.

If you keep the Interpreted UDFs panel open while you are in the process of debugging your UDF, the Interpret button can be used repeatedly since you can make changes with an editor in a separate window. Then, you can continue to debug and interpret until no errors are reported. Remember to save changes to your source code file in the editor window before trying to interpret again.

One of the more common errors made when interpreting source files is trying to interpret source code that contains elements of C that the interpreter does not accommodate. For example, if you have code that contains a structured reference call (which is not supported by the C preprocessor), the interpretation will fail and you will get an error message similar to the following:

```
Error: /nsf/erflnx/home/erf/fluent/udfexample.c:
line 15: structure reference
```

See Section 1.3 for a list of limitations of interpreted UDFs.
7.3 Compiling a Source File Using the Compiled UDFs Panel

Special Considerations for Parallel FLUENT

If you are running the parallel version of FLUENT on a Windows network and you encounter errors when trying to interpret your source file(s), it could be the result of an improper installation of cpp. Proper installation of parallel FLUENT for Windows ensures that the FLUENT INC environment variable is set to the shared directory where FLUENT is installed. If the variable is defined locally instead, the following error message will be reported when you try to interpret a source file:

**Warning: unable to run cpp**

You will need to see your system administrator to reset the FLUENT INC environment variable.

7.3 Compiling a Source File Using the Compiled UDFs Panel

The process of compiling source file(s) involves two steps: build and load. First a shared library is built for the source file(s), and then the library is loaded into FLUENT. Once loaded, the compiled UDFs in the shared library will appear in drop-down lists in graphical user interface panels in FLUENT. You can also write a case file once the functions are loaded so that the compiled UDFs will be stored along with the case, and will be automatically loaded upon subsequent reads.

The general procedure for compiling source file(s) is as follows:

1. Make sure that the UDF source file(s) are in the same directory that contains your case and data files.

   If you are running the parallel version of FLUENT on a network of Windows machines, you must ‘share’ the working directory that contains your UDF source, case, and data files so that all of the compute nodes in the cluster can see it. To do this:

   (a) Open the Windows Explorer application, right click on the folder for the working directory (e.g., mywork), select the Sharing... option, and specify a Share Name (e.g., mywork).

2. Start FLUENT from your working directory.

3. Read (or set up) your case file.

4. Open the Compiled UDFs panel (Figure 7.3.1).

   Define → User-Defined → Functions → Compiled...

5. Select the UDF source file(s) you want to compile by clicking Add... under Source Files in the Compiled UDFs panel (Figure 7.3.1). This will open the Select File panel.
Interpreting and Compiling Source Files

Figure 7.3.1: The Compiled UDFs Panel

Figure 7.3.2: The Select File Panel
7.3 Compiling a Source File Using the Compiled UDFs Panel

(a) In the Select File panel (Figure 7.3.2), highlight the directory path under Directories, and the desired file (e.g., udfexample.c) under Files. Once highlighted, the complete path to the source file will be displayed under Source File(s). Click OK.

Repeat this step until all source files are selected.

! If you are running FLUENT on a network of Windows machines, you may need to type the file’s complete path in the Source File Name field in the Interpreted UDFs panel, instead of using the browser option. For example, to compile udfexample.c from a shared working directory named mywork, you would enter the following in the Source File Name field:

```
\\<fileserver>\mywork\udfexample.c
```

replacing <fileserver> with the name of the computer on which your working directory (mywork) and source file (udfexample.c) are located.

6. Select the UDF header file(s) that are required for compilation by clicking Add... under Header File(s) in the Compiled UDFs panel (Figure 7.3.1). This will open the Select File panel.

(a) In the Select File panel (Figure 7.3.2), highlight the directory path under Directories, and the desired file under Files. Once highlighted, the complete path to the header file will be displayed under Header File(s). Click OK.

Repeat this step until all header files are selected.

! If you are running FLUENT on a network of Windows machines, you may need to type the file’s complete path in the Header File Name field, instead of using the browser option. See the previous step for details.

7. Build a shared library for your source file(s) by typing a name for your shared library next to Library Name in the Compiled UDFs panel (Figure 7.3.1), or leave the default name libudf, and click Build.

As the build process begins, an Information dialog box will appear (Figure 7.3.3) and the results of the build will be displayed on the console window. Click OK to close the dialog box. You can view the compilation history in the ‘log’ file that is saved in your working directory.

! If your build is unsuccessful, then FLUENT will report an error and you will need to debug your program before continuing.

8. Load the shared library you just built into FLUENT by clicking Load unless you are using the parallel version of FLUENT on a Windows network (see below). A message will be displayed on the console window providing a status of the load process. For example:
Opening library "libudf"...
Library "libudf/hpux11/2d/libudf.so" opened
   inlet_x_velocity
Done.

If you are running the parallel version of FLUENT on a network of Windows machines, do the following:

(a) In the Compiled UDFs panel, type
   \\<fileserver>\mywork\libudf
in the Library Name field, replacing \<fileserver> with the name of the computer on which your working directory (\mywork), and shared library directory (\libudf), are located. Click Load.

9. Close the Compiled UDFs panel when your library is successfully loaded. See Section 7.3.1 if you encounter an error when loading.

10. Write the case file if you want the compiled function(s) in the shared library to be saved with the case, and automatically loaded into FLUENT when the case is subsequently read.

   If you do not want the shared library saved with your case file, then you must remember to load it into FLUENT using the Compiled UDFs panel in subsequent sessions.
7.3 Compiling a Source File Using the Compiled UDFs Panel

7.3.1 Common Errors Made While Loading Your Compiled Library

You will get an error message when you try to read a case file that was originally saved with a shared library that you have subsequently moved to a different location.

Opening library "libudf...
Error: open_udf_library: couldn’t open library: libudf/ln86/2d/libudf.so

You will get an error message when you try to load a shared library before it has been built.

Opening library "libudf...
Error: open_udf_library: No such file or directory

Special Considerations for Parallel FLUENT

If you are running the parallel version of FLUENT on a Windows network, the directory containing the shared library (e.g., sharedlibudf) must be shared with all of the remote nodes in the cluster. If you forget to enable the sharing option for the library directory using the Windows Explorer (see Step 8 of the previous section), then FLUENT will hang when you try to load the library (e.g., sharedlibudf) in the Compiled UDFs panel.
Chapter 8. Hooking Your UDF to FLUENT

Once you have interpreted or compiled your UDF using one of the methods described in Chapter 7, you are ready to hook the function and use it in your FLUENT model. Details about hooking your UDF to FLUENT can be found in the following sections. Note that these sections relate to corresponding DEFINE macro sections in Chapter 4.

- Section 8.1: Hooking General Solver UDFs to FLUENT
- Section 8.2: Hooking Model-Specific UDFs to FLUENT
- Section 8.3: Hooking Multiphase UDFs to FLUENT
- Section 8.4: Hooking Dynamic Mesh UDFs to FLUENT
- Section 8.5: Hooking DPM UDFs to FLUENT
8.1 Hooking General Solver UDFs to FLUENT

This section contains methods for hooking UDFs to FLUENT that have been

- defined using DEFINE macros described in Section 4.2, and
- interpreted or compiled using methods described in Chapter 7.

8.1.1 DEFINE_ADJUST

Once you have interpreted or compiled your DEFINE_ADJUST UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.1.1) in FLUENT.

![Figure 8.1.1: The User-Defined Function Hooks Panel](image)

To hook the UDF to FLUENT, choose the function name (e.g., my_adjust) in the Adjust Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.2.1 for details about DEFINE_ADJUST functions.
8.1.2 DEFINE_EXECUTE_AT_END

Once you have interpreted or compiled your DEFINE_EXECUTE_AT_END UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.1.2) in FLUENT.

Define → User-Defined → Function Hooks...

![User-Defined Function Hooks Panel](image)

Figure 8.1.2: The User-Defined Function Hooks Panel

To hook the UDF to FLUENT, choose the function name (e.g., execute_at_end) in the Execute At End Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.2.2 for details about DEFINE_EXECUTE_AT_END functions.
8.1.3 DEFINE_INIT

Once you have interpreted or compiled your DEFINE_INIT UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.1.3) in FLUENT.

Define → User-Defined → Function Hooks...

![User-Defined Function Hooks Panel](image)

Figure 8.1.3: The User-Defined Function Hooks Panel

To hook the UDF to FLUENT, choose the function name (e.g., my_init_function) in the Initialization Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.2.3 for details about DEFINE_INIT functions.
8.1.4 DEFINE_ON_DEMAND

Once you have interpreted or compiled your DEFINE_ON_DEMAND UDF (see Chapter 7), the name of the function will become visible and selectable in the Execute UDF On Demand panel (Figure 8.1.4) in FLUENT.

![Execute On Demand Panel](image)

Figure 8.1.4: The Execute On Demand Panel

To hook the UDF to FLUENT, choose the function name (e.g., update) in the Function drop-down list in the Execute On Demand panel, and click Execute and then Close. FLUENT will execute the UDF immediately. Click Close to close the panel.

See Section 4.2.4 for details about DEFINE_ON_DEMAND functions.
8.1.5 DEFINE_RW_FILE

Once you have interpreted or compiled your DEFINE_RW_FILE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.1.5) in FLUENT.

![User-Defined Function Hooks Panel](image)

To hook the UDF to FLUENT, choose the function name (e.g., reader) in one of the appropriate Function drop-down lists (e.g., Read Case Function) in the User-Defined Function Hooks panel, and click OK. (Figure 8.1.5) Below is a description of each RW File function option.

- **Read Case Function** is called when you read a case file into FLUENT. It will specify the customized section that is to be read from the case file.

- **Write Case Function** is called when you write a case file from FLUENT. It will specify the customized section that is to be written to the case file.

- **Read Data Function** is called when you read a data file into FLUENT. It will specify the customized section that is to be read from the data file.

- **Write Data Function** is called when you write a data file from FLUENT. It will specify the customized section that is to be written to the data file.

See Section 4.2.5 for details about DEFINE_RW_FILE functions.
8.1 Hooking General Solver UDFs to FLUENT

8.1.6 User-Defined Memory Storage

You can store values computed by your UDF in memory so that they can be retrieved later, either by a UDF or for postprocessing within FLUENT. In order to have access to this memory, you will need to specify the Number of User-Defined Memory Locations in the User-Defined Memory panel (Figure 8.1.6).

Define → User-Defined → Memory...

Figure 8.1.6: The User-Defined Memory Panel

The macro C_UDMI or F_UDMI can be used in your UDF to access a particular user-defined memory location in a cell or face, respectively. See Sections 5.3 and 5.4 for details.

Field values that have been stored in user-defined memory will be saved to the data file when you next write one. These fields will also appear in the User Defined Memory... category in the drop-down lists in FLUENT’s postprocessing panels. They will be named udm-0, udm-1, etc., based on the memory location index. The total number of memory locations is limited to 500. For large numbers of user-defined memory locations, system memory requirements will increase.
8.2 Hooking Model-Specific UDFs to FLUENT

This section contains methods for hooking UDFs to FLUENT that have been

- defined using DEFINE macros found in Section 4.3, and
- interpreted or compiled using methods described in Chapter 7.

8.2.1 DEFINE_CHEM_STEP

Once you have interpreted or compiled your DEFINE_CHEM_STEP UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.1) in FLUENT.

![User-Defined Function Hooks Panel](image)

Figure 8.2.1: The User-Defined Function Hooks Panel

EDC or PDF Transport models must be enabled to hook chemistry step UDFs.

To hook the UDF to FLUENT, choose the function name (e.g., user_chem_step) in the Chemistry Step Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.1 for details about DEFINE_CHEM_STEP functions.
8.2.2 DEFINE_DELTAT

Once you have interpreted or compiled your DEFINE_DELTAT UDF (see Chapter 7), the name of the function will become visible and selectable in the Iterate panel (Figure 8.2.2) in FLUENT.

To hook the UDF to FLUENT, you will first need to select Adaptive as the Time Stepping Method in the Iterate panel, and then choose the function name (e.g., mydeltat) in the User-Defined Time Step drop-down list under Adaptive Time Stepping.

See Section 4.3.2 for details about DEFINE_DELTAT functions.
8.2.3 DEFINE_DIFFUSIVITY

Once you have interpreted or compiled your DEFINE_DIFFUSIVITY UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Functions panel (Figure 8.2.4) in FLUENT. To hook the UDF to FLUENT, you will first need to open the User-Defined Functions panel by choosing user-defined in the drop-down list for the Mass Diffusivity property in the Materials panel (Figure 8.2.3).

![Material Panel](image)

Figure 8.2.3: The Materials Panel
Next, choose the function name (e.g., `mean_age_diff`) from the list of UDFs in the User-Defined Functions panel, (Figure 8.2.4) and click OK. The name of the function will subsequently be displayed under the Mass Diffusivity property in the Materials panel.

![User-Defined Functions Panel](image.png)

Figure 8.2.4: The User-Defined Functions Panel

See Section 4.3.3 for details about `DEFINE_DIFFUSIVITY` functions.
8.2.4 DEFINE_DOM_DIFFUSE_REFLECTIVITY

Once you have interpreted or compiled your DEFINE_DOM_DIFFUSE_REFLECTIVITY UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.5) in FLUENT.

![User-Defined Function Hooks Panel]

Figure 8.2.5: The User-Defined Function Hooks Panel

The Discrete Ordinates radiation model must be enabled from the Radiation Model panel. To hook the UDF to FLUENT, choose the function name (e.g., user_dom_diff_refl) in the DO Diffuse Reflectivity Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.4 for details about DEFINE_DOM.Diffuse.Reflectivity functions.
8.2 Hooking Model-Specific UDFs to FLUENT

8.2.5 DEFINE_DOM_SOURCE

Once you have interpreted or compiled your DEFINE_DOM_SOURCE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.6) in FLUENT.

The Discrete Ordinates radiation model must be enabled.

![User-Defined Function Hooks Panel](image)

Figure 8.2.6: The User-Defined Function Hooks Panel

To hook the UDF to FLUENT, choose the function name (e.g., user_dom_source) in the DO Source Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.5 for details about DEFINE_DOM_SOURCE functions.
8.2.6 DEFINE_DOM_SPECULAR_REFLECTIVITY

Once you have interpreted or compiled your DEFINE_DOM_SPECULAR_REFLECTIVITY UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.7) in FLUENT.

![User-Defined Function Hooks Panel](image)

Figure 8.2.7: The User-Defined Function Hooks Panel

To hook the UDF to FLUENT, choose the function name (e.g., `user_dom_spec_ref1`) in the DO Specular Reflectivity Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.4 for details about DEFINE_DOM_SPECULAR_REFLECTIVITY functions.
8.2.7 DEFINE_HEAT_FLUX

Once you have interpreted, or compiled and linked your DEFINE_HEAT_FLUX UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.8) in FLUENT.

Define → User-Defined → Function Hooks...

![User-Defined Function Hooks Panel](image)

Figure 8.2.8: The User-Defined Function Hooks Panel

! The Energy Equation must be enabled.

To hook the UDF to FLUENT, simply choose the function name (e.g., heat_flux) in the Wall Heat Flux Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.7 for details about DEFINE_HEAT_FLUX functions.
8.2.8 DEFINE_NET_REACTION_RATE

Once you have interpreted or compiled your DEFINE_NET_REACTION_RATE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.9) in FLUENT.

![User-Defined Function Hooks Panel](image)

Figure 8.2.9: The User-Defined Function Hooks Panel

Net reaction rate UDFs may be used for the EDC and PDF Transport models, as well as for the surface chemistry model. To enable the PDF Transport models, select Composition PDF Transport and Volumetric reactions in the Species Model panel. To enable the EDC model, select Species Transport and Volumetric reactions in the Species Model panel, and choose EDC under Turbulence-Chemistry Interaction.

To hook the UDF to FLUENT, choose the function name (e.g., `user_net_reaction_rate`) in the Net Reaction Rate Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.8 for details about DEFINE_NET_REACTION_RATE functions.
8.2.9 DEFINE_NOX_RATE

Once you have interpreted or compiled your `DEFINE_NOX_RATE` UDF (see Chapter 7), the name of the function will become visible and selectable in the NOx Model panel (Figure 8.2.10) in FLUENT.

To hook the UDF to FLUENT, choose the function name (e.g., `nox_rate`) in the NOX Rate drop-down list under User-Defined Functions in the NOx Model panel, and click OK.

See Section 4.3.9 for details about `DEFINE_NOX_RATE` functions.
8.2.10 DEFINE_PR_RATE

Once you have interpreted or compiled your DEFINE_PR_RATE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.11) in FLUENT.

![Define → User-Defined → Function Hooks...](image)

Figure 8.2.11: The User-Defined Function Hooks Panel

You must enable the particle surface reactions option before you can hook the UDF by selecting Volumetric and Particle Surface under Reactions in the Species Model panel.

To hook the UDF to FLUENT, choose the function name (e.g., user_pr_rate) in the Particle Reaction Rate Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.10 for details about DEFINE_PR_RATE functions.
8.2 Hooking Model-Specific UDFs to FLUENT

8.2.11 DEFINE_PRANDTL

Once you have interpreted or compiled your DEFINE_PRANDTL UDF (see Chapter 7), the name of the function will become visible and selectable in the Viscous Model panel (Figure 8.2.12) in FLUENT.

Define → Models → Viscous...

Figure 8.2.12: The Viscous Model Panel

To hook the UDF to FLUENT, choose the function name (e.g., user_pr_k) in the TKE Prandtl Number drop-down list under User-Defined Functions in the panel Viscous Model panel, and click OK.

See Section 4.3.11 for details about DEFINE_PRANDTL functions.
8.2.12 DEFINE_PROFILE

Once you have interpreted or compiled your DEFINE_PROFILE UDF (see Chapter 7), the name of the function will become visible and selectable in the appropriate boundary condition panel in FLUENT.

Define → Boundary Conditions...

If, for example, your UDF defines a velocity inlet boundary condition, then to hook it to FLUENT, choose the function name (e.g., inlet.x.velocity) in the appropriate drop-down list (e.g., X Velocity) in the Velocity Inlet panel, (Figure 8.2.13) and click OK. Note that the UDF name that is displayed in the drop-down lists is preceded by the word udf (e.g., udf inlet_x_velocity).

![Velocity Inlet Panel](image)

Figure 8.2.13: The Velocity Inlet Panel

If you are using your UDF to specify a fixed value in a cell zone, you will need to turn on the Fixed Values option and select the name of your UDF in the appropriate drop-down list in the Fluid or Solid panel.

See Section 4.3.12 for details about DEFINE_PROFILE functions.
8.2.13 DEFINE_PROPERTY

Material Properties

Once you have interpreted or compiled your material property UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Functions panel (Figure 8.2.15) in FLUENT. To hook the UDF to FLUENT, you will first need to open the User-Defined Functions panel by choosing user-defined in the drop-down list for the appropriate property (e.g., Viscosity) in the Materials panel (Figure 8.2.14).

![Materials Panel](image)

Figure 8.2.14: The Materials Panel
Next, choose the function name (e.g., cell_viscosity) from the list of UDFs displayed in the User-Defined Functions panel, (Figure 8.2.15) and click OK. The name of the function will subsequently be displayed under the selected property (e.g., Viscosity) in the Materials panel.

! If you plan to define density using a UDF, note that the solution convergence will become poor as the density variation becomes large. Specifying a compressible law (density as a function of pressure) or multiphase behavior (spatially varying density) may lead to divergence. It is recommended that you restrict the use of UDFs for density to weakly compressible flows with mild density variations.

See Section 4.3.13 for details about DEFINE_PROPERTY functions.
8.2 Hooking Model-Specific UDFs to FLUENT

8.2.14 DEFINE_SCAT_PHASE_FUNC

Once you have interpreted or compiled your DEFINE_SCAT_PHASE_FUNC UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Functions panel (Figure 8.2.17) in FLUENT. To hook the UDF to FLUENT, you will first need to open the User-Defined Functions panel from the Material panel by selecting user-defined in the drop-down list for the Scattering Phase Function property (Figure 8.2.16).

![Materials Panel](image)

Figure 8.2.16: The Materials Panel

The Discrete Ordinates radiation model must be enabled from the Radiation Model panel.
Next, choose the function name (e.g., `ScatPhiB2`) from the list of UDFs displayed in the User-Defined Functions panel, and click OK. The name of the function will subsequently be displayed under the Scattering Phase Function property in the Materials panel.

Figure 8.2.17: The User-Defined Functions Panel

See Section 4.3.14 for details about `DEFINE_SCAT_PHASE_FUNC` functions.
8.2.15 DEFINE_SOURCE

Once you have interpreted or compiled your DEFINE_SOURCE UDF (see Chapter 7), the name of the function will become visible and selectable in the Fluid or Solid panel in FLUENT. To hook the UDF to FLUENT, you will first need to turn on the Source Terms option in the Fluid or Solid panel (Figure 8.2.18). (This will expand the panel to display source terms (mass, momentum, etc.)

![Define Boundary Conditions...]](image)

Next, choose the function name (e.g., cell\_x\_source) in the appropriate drop-down list (e.g., X Momentum) in the Fluid or Solid panel, and click OK. Note that the UDF name that is displayed in the drop-down lists is preceded by the word udf (e.g., udf cell\_x\_source).

Figure 8.2.18: The Fluid Panel

See Section 4.3.15 for details about DEFINE_SOURCE functions.
8.2.16 DEFINE_SR_RATE

Once you have interpreted or compiled your DEFINE_SR_RATE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.19) in FLUENT.

![Figure 8.2.19: The User-Defined Function Hooks Panel](image)

You must enable the wall surface reactions option before you can hook the UDF by selecting Volumetric and Wall Surface under Reactions in the Species Model panel.

To hook the UDF to FLUENT, choose the function name (e.g., my_rate) in the Surface Reaction Rate Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.16 for details about DEFINE_SR_RATE functions.
8.2 Hooking Model-Specific UDFs to FLUENT

8.2.17 DEFINE_TURB_PREMIX_SOURCE

Once you have interpreted or compiled your DEFINE_TURB_PREMIX_SOURCE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.20) in FLUENT.

Define → User-Defined → Function Hooks...

! You must have a premixed combustion model enabled in the Species Model panel.

To hook the UDF to FLUENT, choose the function name (e.g., turb_premix_source) in the Turbulent Premixed Source Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.17 for details about DEFINE_TURB_PREMIX_SOURCE functions.
8.2.18 DEFINE_TURBULENT_VISCOSITY

Once you have interpreted or compiled your DEFINE_TURBULENT_VISCOSITY UDF (see Chapter 7), the name of the function will become visible and selectable in the Viscous Model panel (Figure 8.2.21) in FLUENT.

![Viscous Model Panel](image)

Figure 8.2.21: The Viscous Model Panel

To hook the UDF to FLUENT, choose the function name (e.g., user_mu_t) in the Turbulence Viscosity drop-down list under User-Defined Functions in the Viscous Model panel, and click OK.

See Section 4.3.18 for details about DEFINE_TURBULENT_VISCOSITY functions.
8.2 Hooking Model-Specific UDFs to FLUENT

8.2.19 DEFINE_UDS_FLUX

Once you have interpreted or compiled your DEFINE_UDS_FLUX UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Scalars panel (Figure 8.2.22) in FLUENT.

![User-Defined Scalars Panel](image)

Figure 8.2.22: The User-Defined Scalars Panel

To hook the UDF to FLUENT, first specify the Number of User-Defined Scalars in the User-Defined Scalars panel (Figure 8.2.22). As you enter the number of user-defined scalars, the panel will expand to show the Flux Function settings. Next, choose the function name (e.g., my_uds_flux) in the Flux Function drop-down list, and click OK.

See Section 4.3.19 for details about DEFINE_UDS_FLUX functions.
8.2.20 DEFINE_UDS_UNSTEADY

Once you have interpreted or compiled your DEFINE_UDS_UNSTEADY UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Scalars panel (Figure 8.2.23) in FLUENT.

Define → User-Defined → Scalars...

Figure 8.2.23: The User-Defined Scalars Panel

To hook the UDF to FLUENT, first specify the Number of User-Defined Scalars in the User-Defined Scalars panel (Figure 8.2.23). As you enter the number of user-defined scalars, the panel will expand to show the Unsteady Function settings. Next, choose the function name (e.g., uns_time) in the Unsteady Function drop-down list, and click OK.

See Section 4.3.20 for details about DEFINE_UDS_UNSTEADY functions.
8.2 Hooking Model-Specific UDFs to FLUENT

8.2.21 DEFINE VR_RATE

Once you have interpreted or compiled your DEFINE VR_RATE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.2.24) in FLUENT.

You must turn on the volumetric reactions option before you can hook the UDF by selecting Volumetric under Reactions in the Species Model panel.

To hook the UDF to FLUENT, choose the function name (e.g., vol_reac_rate) in the Volume Reaction Rate Function drop-down list in the User-Defined Function Hooks panel, and click OK.

See Section 4.3.21 for details about DEFINE VR_RATE functions.

Figure 8.2.24: The User-Defined Function Hooks Panel
8.3 Hooking Multiphase UDFs to FLUENT

This section contains methods for hooking UDFs to FLUENT that have been

- defined using DEFINE macros described in Section 4.4, and
- interpreted or compiled using methods described in Chapter 7.

8.3.1 DEFINE_CAVITATION_RATE

Once you have interpreted or compiled your DEFINE_CAVITATION_RATE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.3.2) in FLUENT. Note that cavitation rate UDFs can be applied only to the Mixture multiphase model.

To hook the UDF to FLUENT, you will first need to enable the Mixture model in the Multiphase Model panel, and then select Cavitation under Interphase Mass Transfer.

Next, open the User-Defined Function Hooks panel,

Define → Models → Multiphase...

and choose the function name (e.g., user_cavitation_rate) in the Cavitation Mass Rate Function drop-down list (Figure 8.3.2), and click OK.

See Section 4.4.1 for details about DEFINE_CAVITATION_RATE functions.
8.3 Hooking Multiphase UDFs to FLUENT

Figure 8.3.1: The Multiphase Model Panel

Figure 8.3.2: The User-Defined Function Hooks Panel
8.3.2 DEFINE_EXCHANGE_PROPERTY

Once you have interpreted or compiled your DEFINE_EXCHANGE_RATE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Functions panel (Figure 8.3.4) in FLUENT.

Customized mass transfer UDFs can be applied to VOF, Mixture, and Eulerian multiphase models. Drag coefficient UDFs can be applied to Mixture and Eulerian models, while heat transfer and lift coefficient UDFs can be applied only to the Eulerian model. You will need to have the multiphase model enabled before you can hook your function.

To hook an exchange property UDF to FLUENT, you will first need to open the Phase Interaction panel (Figure 8.3.3) by clicking Interactions... in the Phases panel.

Next, click on the appropriate tab (e.g., Drag) in the Phase Interaction panel, and choose user-defined in the drop-down list for the corresponding exchange property (e.g., Drag Coefficient) that you desire. This will open the User-Defined Functions panel.

![Figure 8.3.3: The Phase Interaction Panel](image)

Make sure that you select Slip Velocity under Mixture Parameters in the Multiphase Model panel in order to display the drag coefficient for the Mixture model.
Finally, choose the function name (e.g., drag) from the list of UDFs displayed in the User-Defined Functions panel, (Figure 8.3.4) and click OK. The function name (e.g., drag) will then be displayed under the user-defined function for Drag Coefficient in the Phase Interaction panel.

See Section 4.4.2 for details about DEFINE_EXCHANGE_PROPERTY functions.
8.3.3 DEFINE_VECTOR_EXCHANGE_PROPERTY

Once you have interpreted or compiled your DEFINE_VECTOR_EXCHANGE_RATE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Functions panel (Figure 8.3.6) in FLUENT.

To hook the UDF to FLUENT, you will first need to open the Phase Interaction panel (Figure 8.3.5) by clicking Interactions... in the Phases panel.

![Phase Interaction Panel](image)

Figure 8.3.5: The Phase Interaction Panel

Next, click on the Slip tab in the Phase Interaction panel, and choose user-defined in the drop-down list for the Slip Velocity. This will open the User-Defined Functions panel.

! Slip velocity UDFs apply only to the multiphase Mixture model.
Finally, choose the function name (e.g., custom_slip) from the list of UDFs displayed in the User-Defined Functions panel, (Figure 8.3.4) and click OK.

See Section 4.4.3 for details about DEFINE_VECTOR_EXCHANGEPROPERTY functions.
8.4 Hooking Dynamic Mesh UDFs to FLUENT

This section contains methods for hooking UDFs to FLUENT that have been

- defined using DEFINE macros described in Section 4.5, and
- interpreted or compiled using methods described in Chapter 7.

8.4.1 DEFINE.CG.MOTION

Once you have interpreted or compiled your DEFINE.CG.MOTION UDF (see Chapter 7), the name of the function will become visible and selectable in the Dynamic Zones panel (Figure 8.4.1) in FLUENT. To hook the UDF to FLUENT, you will first need to enable the dynamic mesh model.

Define → Dynamic Mesh → Parameters...

Select Dynamic Mesh under Model and click OK.

The Dynamic Mesh panel will be accessible only when you choose Unsteady as the time method in the Solver panel.

Next, open the Dynamic Zones panel.

Define → Dynamic Mesh → Zones...
Select **Rigid Body** under **Type** in the **Dynamic Zones** panel (Figure 8.4.1) and click on the **Motion Attributes** tab. Finally, choose the function name (e.g., piston) from the **Motion UDF/Profile** drop-down list, and click **Create** then **Close**.

See Section 4.5.1 for details about **DEFINE_CG_MOTION** functions.
8.4.2 DEFINE_GEOm

Once you have interpreted or compiled your DEFINE_GEOm UDF (see Chapter 7), the name of the function will become visible and selectable in the Dynamic Zones panel (Figure 8.4.2) in FLUENT. To hook the UDF to FLUENT, you will first need to enable the Dynamic Mesh model.

Define → Dynamic Mesh → Parameters...

Select Dynamic Mesh under Model and click OK.

The Dynamic Mesh panel will be accessible only when you choose Unsteady as the time method in the Solver panel.

Next, open the Dynamic Zones panel.

Define → Dynamic Mesh → Zones...

Figure 8.4.2: The Dynamic Zones Panel
Select Deforming under Type in the Dynamic Zones panel (Figure 8.4.2) and click on the Geometry Definition tab. Select user-defined in the drop-down list under Definition, and choose the function name (e.g., parabola) from the Geometry UDF drop-down list. Click Create and then Close.

See Section 4.5.2 for details about DEFINE_GEOM functions.
8.4.3 DEFINE_GRID_MOTION

Once you have interpreted or compiled your DEFINE_GRID_MOTION UDF (see Chapter 7), the name of the function will become visible and selectable in the Dynamic Zones panel (Figure 8.4.3) in FLUENT. To hook the UDF to FLUENT, you will first need to enable the Dynamic Mesh model.

Define → Dynamic Mesh → Parameters...

Select Dynamic Mesh under Model and click OK.

The Dynamic Mesh panel will be accessible only when you choose Unsteady as the time method in the Solver panel.

Next, open the Dynamic Zones panel.

Figure 8.4.3: The Dynamic Zones Panel
Select User-Defined under Type in the Dynamic Zones panel (Figure 8.4.3) and click on the Motion Attributes tab. Choose the function name (e.g., beam) from the Mesh Motion UDF drop-down list. Click Create then Close.

See Section 4.5.3 for details about DEFINE_GRID_MOTION functions.
8.5 Hooking DPM UDFs to FLUENT

This section contains methods for hooking UDFs to FLUENT that have been

- defined using DEFINE macros described in Section 4.6, and
- interpreted or compiled using methods described in Chapter 7.

8.5.1 DEFINE_DPM_BC

Once you have interpreted or compiled your DEFINE_DPM_BC UDF (see Chapter 7), the name of the function will become visible and selectable in the appropriate boundary condition panel (Figure 8.5.1) in FLUENT.

Suppose that your UDF defines a particle velocity boundary condition at a wall. To hook the UDF to FLUENT, first open the Wall boundary condition panel and select the DPM tab (Figure 8.5.1).
Then, in the Wall panel (in the DPM tab), choose user_defined as Boundary Cond. Type under Discrete Phase Model Conditions. This will expand the panel to allow you to choose the function name (e.g., reflect) from the Boundary Cond. Function drop-down list. Click OK.

See Section 4.6.1 for details about DEFINE_DPM_BC functions.
8.5.2 DEFINE_DPM_BODY_FORCE

Once you have interpreted or compiled your DEFINE_DPM_BODY_FORCE UDF (see Chapter 7), the name of the function will become visible and selectable in the Discrete Phase Model panel (Figure 8.5.2) in FLUENT.

Define → Models → Discrete Phase...

Figure 8.5.2: The Discrete Phase Model Panel

To hook the UDF to FLUENT, choose the function name (e.g., particle_body_force) in the Body Force drop-down list under User-Defined Functions, (Figure 8.5.2) and click OK. See Section 4.6.2 for details about DEFINE_DPM_BODY_FORCE functions.
8.5 Hooking DPM UDFs to FLUENT

8.5.3 DEFINE_DPM_DRAG

Once you have interpreted or compiled your DEFINE_DPM_DRAG UDF (see Chapter 7), the name of the function will become visible and selectable in the Discrete Phase Model panel (Figure 8.5.3) in FLUENT.

![Discrete Phase Model Panel](image)

Figure 8.5.3: The Discrete Phase Model Panel

To hook the UDF to FLUENT, choose the function name (e.g., particle_drag_force) in the Drag Law drop-down list under Drag Parameters (Figure 8.5.3), and click OK. (Note, function names listed in the drop-down list are preceded by the word udf as in udf particle_drag_force.)

See Section 4.6.3 for details about DEFINE_DPM_DRAG functions.
8.5.4 DEFINE_DPM_EROSION

Once you have interpreted or compiled your DEFINE_DPM_EROSION UDF, the name of the function will become visible and selectable in the Discrete Phase Model panel (Figure 8.5.4) in FLUENT.

Define — Models — Discrete Phase...

Figure 8.5.4: The Discrete Phase Model Panel
To hook the UDF to FLUENT, enable the Interaction with Continuous Phase option under Interaction (Figure 8.5.4), and then turn on Erosion/Accretion under Option. Finally, choose the function name (e.g., dpm_accr) in the Erosion/Accretion drop-down list under User-Defined Functions, and click OK.

See Section 4.6.4 for details about DEFINE_DPM_EROSION functions.
8.5.5 DEFINE_DPM_INJECTION_INIT

Once you have interpreted or compiled your DEFINE_DPM_INJECTION_INIT UDF (see Chapter 7), the name of the function will become visible and selectable in the Set Injection Properties panel (Figure 8.5.5) in FLUENT. Before you hook the UDF, you’ll need to create your particle injections in the Injections panel.

Define —> Injections...

Click Create in the Injections panel to open the Set Injection Properties panel and set up your particle injections.

Next, select the UDF tab in the Set Injection Properties panel (Figure 8.5.5), and choose the function name (e.g., init_bubbles) from the Initialization drop-down list under User-Defined Functions. Click OK.
See Section 4.6.5 for details about DEFINE_DPM_INJECTION_INIT functions.

8.5.6 DEFINE_DPM_LAW

Once you have interpreted or compiled your DEFINE_DPM_LAW UDF (see Chapter 7), the name of the function will become visible and selectable in the Custom Laws panel (Figure 8.5.6) in FLUENT. To hook the UDF to FLUENT, first click Create in the Injections panel to open the Set Injection Properties panel.

Next, turn on the Custom option under Laws in the Set Injection Properties panel. This will open the Custom Laws panel.

![Custom Laws Panel](image)

Figure 8.5.6: The Custom Laws Panel

Finally, in the Custom Laws panel, (Figure 8.5.6) choose the function name (e.g., custom_law) in the appropriate drop-down list located to the left of each of the six particle laws (e.g., First Law), and click OK.

See Section 4.6.6 for details about DEFINE_DPM_LAW functions.
8.5.7  DEFINE_DPM_OUTPUT

Once you have interpreted or compiled your DEFINE_DPM_OUTPUT UDF (see Chapter 7), the name of the function will become visible and selectable in the Sample Trajectories panel (Figure 8.5.7) in FLUENT.

To hook the UDF to FLUENT, choose the function name (e.g., dpm_output) in the Output drop-down list under User-Defined Functions, and click Compute and Close.

See Section 4.6.7 for details about DEFINE_DPM_OUTPUT functions.
8.5.8 DEFINE_DPM_PROPERTY

Once you have interpreted or compiled your DEFINE_DPM_PROPERTY UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Functions panel (Figure 8.5.9) in FLUENT. To hook the UDF to FLUENT, you will first need to open the User-Defined Functions panel by choosing user-defined in the drop-down list for the appropriate property (e.g., Particle Emissivity) in the Materials panel (Figure 8.5.8).

![Materials Panel](image)

Figure 8.5.8: The Materials Panel

In order for the Particle Emissivity property to be displayed in the sample panel shown above, you must enable a radiation model, turn on the Particle Radiation Interaction option in the Discrete Phase Model panel, and introduce a particle injection in the Injections panel.
Next, choose the function name (e.g., coal\_emissivity) from the list of UDFs displayed in the User-Defined Functions panel, (Figure 8.5.9) and click OK. The name of the function will subsequently be displayed under the selected property (e.g., Particle Emissivity) in the Materials panel.

See Section 4.3.13 for details about DEFINE\_DPM\_PROPERTY functions.
8.5.9 DEFINE_DPM_SCALAR_UPDATE

Once you have interpreted or compiled your DEFINE_DPM_SCALAR_UPDATE UDF (see Chapter 7), the name of the function will become visible and selectable in the Discrete Phase Model panel (Figure 8.5.10) in FLUENT.

To hook the UDF to FLUENT, choose the function name (e.g., melting_index) in the Scalar Update drop-down list under User-Defined Functions (Figure 8.5.10), and click OK.

See Section 4.6.9 for details about DEFINE_DPM_SCALAR_UPDATE functions.
8.5.10 DEFINE_DPM_SOURCE

Once you have interpreted or compiled your DEFINE_DPM_SOURCE UDF (see Chapter 7), the name of the function will become visible and selectable in the Discrete Phase Model panel (Figure 8.5.11) in FLUENT.

To hook the UDF to FLUENT, choose the function name (e.g., dpm_source) in the Source drop-down list under User-Defined Functions (Figure 8.5.11), and click OK.

See Section 4.6.10 for details about DEFINE_DPM_SOURCE functions.
8.5.11 DEFINE_DPM_SPRAY_COLLIDE

Once you have interpreted or compiled your DEFINE_DPM_SPRAY_COLLIDE UDF (see Chapter 7), the name of the function will become visible and selectable in the User-Defined Function Hooks panel (Figure 8.5.12) in FLUENT.

Define → User-Defined → Function Hooks...

![User-Defined Function Hooks Panel](image)

Figure 8.5.12: The User-Defined Function Hooks Panel

You will need to enable a discrete phase model in the Discrete Phase Model panel before you can hook the UDF.

To hook the UDF to FLUENT, choose the function name (e.g., udf_mean_spray) in the Spray Collide Function drop-down list in the User-Defined Function Hooks panel, (Figure 8.5.12) and click OK.

See Section 4.6.11 for details about DEFINE_DPM_SPRAY_COLLIDE functions.
8.5.12 DEFINE_DPM SWITCH

Once you have interpreted or compiled your DEFINE_DPM_LAW UDF (see Chapter 7), the name of the function will become visible and selectable in the Custom Laws panel (Figure 8.5.13) in FLUENT. To hook the UDF to FLUENT, first click Create in the Injections panel to open the Set Injection Properties panel.

Next, turn on the Custom option under Laws in the Set Injection Properties panel. This will open the Custom Laws panel.

![Custom Laws Panel]

Finally, in the Custom Laws panel (Figure 8.5.13) choose the function name (e.g., dpm_switch) from the last drop-down list labeled Switching, (Figure 8.5.13) and click OK. See Section 4.6.12 for details about DEFINE_DPM SWITCH functions.
Chapter 9. Parallel UDF Usage

This chapter contains an overview of user-defined functions (UDFs) for parallel FLUENT and their usage. Details about parallel UDF functionality can be found in the following sections:

- Section 9.1: Overview of Parallel FLUENT
- Section 9.2: Cells and Faces in a Partitioned Grid
- Section 9.3: Parallelizing Your Serial UDF
- Section 9.4: Macros for Parallel UDFs
- Section 9.6: Parallel UDF Example
- Section 9.5: Process Identification
- Section 9.7: Writing Files in Parallel

9.1 Overview of Parallel FLUENT

Fluent Inc.’s parallel solver simultaneously computes a solution to a large problem by using multiple processes that may be executing on the same machine, or on different machines in a network. It does this by splitting up the solution grid into multiple partitions (Figure 9.1.1) and assigning each data partition to a different compute process, referred to as a compute node (Figure 9.1.2.) Each compute node executes the same program on its own data set, simultaneously, with every other compute node. The host process, or simply the host, does not contain grid cells, faces, or nodes (except when using the DPM shared-memory model). Its primary purpose is to interpret commands from Cortex (the FLUENT process responsible for user-interface and graphics-related functions) and in turn, to pass those commands (and data) to a compute node which distributes it to the other compute nodes.
Parallel UDF Usage

Figure 9.1.1: Partitioned Grid in Parallel FLUENT

Figure 9.1.2: Partitioned Grid Distributed Between Two Compute Nodes
Figure 9.1.3: Domain and Thread Mirroring in a Distributed Grid

Compute nodes store and perform computations on their portion of the mesh as well as a single layer of copied “overlapping” cells at the edges that join to other partitions (Figure 9.1.2). Even though the cells and faces are partitioned, all of the domains and threads in a grid are mirrored on each compute node (Figure 9.1.3). The threads are stored as linked lists as in the serial solver. The compute nodes can be implemented on a massively parallel computer, a multiple-CPU workstation, or a network of workstations using the same or different operating systems.
9.1.1 Command Transfer and Communication

The processes that are involved in a FLUENT session running in parallel are defined by Cortex, a host process, and a set of n compute node processes (referred to as compute nodes), with compute nodes being labeled from 0 to n-1 (Figure 9.1.4). The host receives commands from Cortex and passes commands to compute node-0. Compute node-0, in turn, sends commands to the other compute nodes. All compute nodes (except 0) receive commands from compute node-0. Before the compute nodes pass messages to the host (via compute node-0), they synchronize with each other. Figure 9.1.4 shows the relationship of processes in parallel FLUENT.

Each compute node is ‘virtually’ connected to every other compute node and relies on its “communicator” to perform such functions as sending and receiving arrays, synchronizing, performing global reductions (such as summations over all cells), and establishing machine connectivity. A FLUENT communicator is a message-passing library. For example, it could be a vendor implementation of the Message Passing Interface (MPI) standard, as depicted in Figure 9.1.4.

All of the parallel FLUENT processes (as well as the serial process) are identified by a unique integer ID. The host process is assigned the ID node_host(=999999). The host collects messages from compute node-0 and performs operation (such as printing, displaying messages, and writing to a file) on all of the data, in the same way as the serial solver. (Figure 9.1.5)
Figure 9.1.4: Parallel FLUENT Architecture
Figure 9.1.5: Example of Command Transfer in Parallel FLUENT
9.2 Cells and Faces in a Partitioned Grid

Some terminology needs to be introduced to distinguish between different types of cells and faces in a partitioned grid. Note that this nomenclature applies only to parallel coding in FLUENT.

Cell Types in a Partitioned Grid

There are two types of cells in a partitioned grid: interior cells and exterior cells (Figure 9.2.1). Interior cells are fully contained within a grid partition. Exterior cells on one compute node correspond to the same interior cells in the adjacent compute node. (Figure 9.1.2). This duplication of cells at a partition boundary becomes important when you want to loop over cells in a parallel grid. There are separate macros for looping over interior cells, exterior cells, and all cells. See Section 9.4.5 for details.
There are three classifications of faces in a partitioned grid: interior, boundary zone, and external (Figure 9.2.2). Interior faces have two neighboring cells. Interior faces that lie on a partition boundary are referred as “partition boundary faces.” Boundary zone faces lie on a physical grid boundary and have only one adjacent cell neighbor. External faces are non-partition boundary faces that belong to exterior cells. External faces are generally not used in parallel UDFs and, therefore, will not be discussed here.
9.2 Cells and Faces in a Partitioned Grid

Note that each partition boundary face is duplicated on adjacent compute nodes (Figure 9.1.2). This is necessary so that each compute node can calculate its own face values. However, this duplication can result in face data being counted twice when UDFs are involved in operations that involve summing data in a thread that contains partition boundary faces. For example, if your UDF is tasked with summing data over all of the faces in a grid, then as each node loops over its faces, duplicated partition boundary faces can be counted twice. For this reason, one compute node in every adjacent set is assigned by FLUENT as the “principal” compute node, with respect to partition boundary faces. In other words, although each face can appear on one or two partitions, it can only “officially” belong to one of them. The boolean macro \texttt{PRINCIPAL\_FACE\_P}(f,t) returns \texttt{TRUE} if the face \texttt{f} is a principal face on the current compute node.

\textbf{PRINCIPAL\_FACE\_P}

You can use \texttt{PRINCIPAL\_FACE\_P} to test whether a given face is the principal face, before including it in a face loop summation. In the example below, the area of a face is only added to the total area if it is the principal face. Note that \texttt{PRINCIPAL\_FACE\_P} is always \texttt{TRUE} for the serial version.

\textbf{Example}

\begin{verbatim}
begin_f_loop(f,t)
  if PRINCIPAL_FACE_P(f,t) /* tests if the face is the principle face */
  {
    F_AREA(area,f,t); /* computes area of each face */
    total_area +=NV_MAG(area); /* computes total face area by accumulating magnitude of the each face’s area */
  }
end_f_loop(f,t)
\end{verbatim}
Exterior Thread Storage

Each thread stores the data associated with its cells or faces in a set of arrays. For example, pressure is stored in an array and the pressure for cell c is obtained by accessing element c of that array. Storage for exterior cell and face data occurs at the end of every thread data array, as shown in Figure 9.2.3.

Figure 9.2.3: Exterior Thread Data Storage at End of a Thread Array
9.3 Parallelizing Your Serial UDF

FLUENT’s serial solver contains Cortex and only a single FLUENT process. The parallel solver, on the other hand, contains three types of executable: Cortex, host, and compute node (or simply “node” for short). When FLUENT runs in parallel, an instance of Cortex starts, followed by one host and n compute nodes, thereby giving a total of n+2 running processes. For this reason, when you are running in parallel, you will need to make sure that your function will successfully execute as a host and a node process. At first it may appear that you should write three different versions of your UDF: one for serial, host, and node. Good programming practice, however, would suggest that you write a single UDF that, when compiled, can execute on any of the three versions. This process is referred to in this manual as “parallelizing” your serial UDF. You can do this by adding special macros for parallel as well as compiler directives to your UDF, as described below.

Compiler directives, (e.g., \#if RP_NODE, RP_HOST, PARALLEL) and their negated forms, direct the compiler to include only portions of the function that apply to a particular process, and ignore the rest (see Section 9.4.1).

A general rule of thumb is that your serial UDF needs to be “parallelized” if it performs an operation that is dependent on sending or receiving data from another compute node (or the host). UDFs that involve global reductions such as global sums, minimums or maximums, or ones that perform computations on data residing in adjacent compute nodes, for example, will need to be modified in order to run in parallel. Some other types of operations that require parallelization of serial source code include the following:

- Reading and Writing Files
- Global Reductions
- Global Sums
- Global Minimums and Maximums
- Global Logicals
- Certain Loops over Cells and Faces
- Displaying Messages on a Console
- Printing to a Host or Node Process

Once the source code for your “parallelized” UDF has been written, it can be compiled using the same methods for serial UDFs. Instructions for compiling UDFs can be found in Chapter 7.
9.4 Macros for Parallel UDFs

This section contains macros that you can use to parallelize your serial UDF. Where applicable, definitions for these macros can be found in the referenced header file (e.g., para.h).

9.4.1 Compiler Directives

When converting a UDF to run in parallel, some parts of the function may need to be done by the host and some by the compute nodes. This distinction is made when the UDF is compiled. By using Fluent-provided compiler directives, you can specify portions of your function to be assigned to the serial process, the host, or to the compute nodes. The UDF that you write will be written as a single file for the serial, parallel host and parallel node versions, but different parts of the function will be compiled to generate different versions of the dynamically linked shared object file libudf.so (libudf.dll on NT/Windows). Print tasks, for example, may be assigned exclusively to the host, while a task such as computing the total volume of a complete mesh will be assigned to the compute nodes. Since most operations are executed by the serial solver and either the host or compute nodes, negated forms of compiler directives are more commonly used.

Note that the primary purpose of the host is to interpret commands from Cortex and to pass those commands (and data) to compute node-0 for distribution. Since the host does not contain grid data, you will need to be careful not include the host in any calculations that could, for example result in a division by zero. In this case, you will need to direct the compiler to ignore the host when it is performing grid-related calculations, by wrapping those operations around the #if !RP_HOST directive. For example, suppose that your UDF will compute the total area of a face thread, and then use that total area to compute a flux. If you do not exclude the host from these operations, the total area on the host will be zero and a segmentation fault will occur when the function attempts to divide by zero to obtain the flux.

Example

```c
#if !HOST
avg_pres = total_pres_a / total_area; /* if you don’t exclude the host
  this operation will result in a division by zero and error!
  Remember that host has no data so its total will be zero.*/
#endif
```

You will need to use the #if !RP_NODE directive when you want to exclude compute nodes from operations for which they do not have data. For example, suppose that you specify a user-defined Scheme variable in the text interface. This variable is then defined only on the host, since the host communicates directly with Cortex. If your function performs an operation on the user-defined Scheme variable (say using RP_GetInteger),
and it does not exclude the compute nodes from the operation (they don’t have access to the variable), then you will get a compiler error.

Below is a list of parallel compiler directives and what they do. Note that if either RP_HOST or RP_NODE are true, then PARALLEL is also true.

```c
#ifndef RP_HOST
  /* only host process is involved */
#endif

#ifndef RP_NODE
  /* only compute nodes are involved */
#endif

#ifndef PARALLEL
  /* both host and compute nodes are involved, but not serial equivalent to #if RP_HOST || RP_NODE */
#endif

#ifndef !RP_HOST
  /* either serial or compute node process is involved */
#endif

#ifndef !RP_NODE
  /* either serial or host process is involved */
#endif

#ifndef !PARALLEL
  /* only serial process is involved */
#endif
```

The following simple UDF shows the use of compiler directives. The adjust function is used to define a function called `where_am_i`. This function queries to determine which type of process is executing and then displays a message on that computed node’s monitor.
Example

/********************************************
Simple UDF that uses compiler directives
*********************************************/
#include "udf.h"
DEFINE_ADJUST(where_am_i, domain)
{
#if RP_HOST
   Message("I am in the host process\n");
#endif /* RP_HOST */

#if RP_NODE
   Message("I am in the node process with ID %d\n",myid);
   /* myid is a global variable which is set to the multiport ID for each node */
#endif /* RP_NODE */

#if !PARALLEL
   Message("I am in the serial process\n");
#endif /* !PARALLEL */
}

This simple allocation of functionality between the different types of process is useful in a limited number of practical situations. For example, you may want to display a message on the compute nodes when a particular computation is being run (by using RP_NODE or !RP_HOST). Or, you can also choose to designate the host process to display messages (by using RP_HOST or !RP_NODE). Usually you want messages written only once by the host process (and the serial process). Simple messages such as “Running the Adjust Function” are straightforward. Alternatively, you may want to collect data from all the nodes and print the total once, from the host. To perform this type of operation your UDF will need some form of communication between processes. The most common mode of communication is between the host and the node processes.
9.4.2 Communicating Between the Host and Node Processes

There are two sets of similar macros that can be used to send data between the host and the compute nodes: host_to_node_type_num and node_to_host_type_num.

Host-to-Node Data Transfer

To send data from the host process to all the node processes (indirectly via compute node-0) we use macros of the form:

host_to_node_type_num(val_1,val_2,...,val_num);

where ‘num’ is the number of variables that will be passed in the argument list and ‘type’ is the data type of the variables that will be passed. The maximum number of variables that can be passed is 7. Arrays and strings can also be passed from host to nodes, one at a time, as shown in the examples below.

Examples

/* integer and real variables passed from host to nodes */
host_to_node_int_1(count);
host_to_node_real_7(len1, len2, width1, width2, breadth1, breadth2, vol);

/* string and array variables passed from host to nodes */
char wall_name[]="wall-17";
int thread_ids[10] = {1,29,5,32,18,2,55,21,72,14};

host_to_node_string(wall_name,8); /* remember terminating NUL character */
host_to_node_int(thread_ids,10);

Note that these host_to_node communication macros do not need to be “protected” by compiler directives for parallel UDFs, because all of these macros automatically do the following:

- send the variable value if compiled as the host version,
- receive and then set the local variable if compiled as a compute node version, and
- do nothing in the serial version.

The most common use for this set of macros is to pass parameters or boundary conditions from the host to the nodes processes. See the example UDF in Section 9.6 for a demonstration of usage.
Node-to-Host Data Transfer

To send data from compute node-0 to the host process we use macros of the form:

\[
\text{node}_\text{to}_\text{host}_\text{type}_\text{num}(\text{val}_1, \text{val}_2, \ldots, \text{val}_\text{num});
\]

where ‘num’ is the number of variables that will be passed in the argument list and ‘type’ is the data type of the variables that will be passed. The maximum number of variables that can be passed is 7. Arrays and strings can also be passed from host to nodes, one at a time, as shown in the examples below.

Note that unlike the \text{host}_\text{to}_\text{node} macros which pass data from the host process to \textit{all} of the compute nodes (indirectly via compute node-0), \text{node}_\text{to}_\text{host} macros pass data \textit{only} from compute node-0 to the host.

Examples

/* integer and real variables passed from compute node-0 to host */
node_to_host_int_1(count);
node_to_host_real_7(len1, len2, width1, width2, breadth1, breadth2, vol);

/* string and array variables passed from compute node-0 to host */
char *string;
int string_length;
real vel[ND_ND];
node_to_host_string(string,string_length);
node_to_host_real(vel,ND_ND);

\text{node}_\text{to}_\text{host} macros do not need to be protected by compiler directives (e.g., \#if RP_NODE) since they automatically do the following:

- send the variable value if the node is compute node-0 and the function is compiled as a node version,
- do nothing if the function is compiled as a node version, but the node is not compute node-0,
- receive and set variables if the function is compiled as the host version, and
- do nothing for the serial version.
The most common usage for this set of macros is to pass global reduction results from compute node-0 to the host process. In cases where the value that is to be passed is computed by all of the compute nodes, there must be some sort of collection (such as a summation) of the data from all the compute nodes onto compute node-0 before the single collected (summed) value can be sent. Refer to the example UDF in Section 9.6 for a demonstration of usage and Section 9.4.4 for a full list of global reduction operations.

### 9.4.3 Predicates

There are a number of macros available in parallel FLUENT that expand to logical tests. These logical macros, referred to as “predicates”, are denoted by the suffix _P_ and can be used as test conditions in your UDF. The following predicates return TRUE if the condition in the parenthesis is met.

```c
/* predicate definitions from para.h header file */

#define MULTIPLE_COMPUTE_NODE_P (compute_node_count > 1)
#define ONE_COMPUTE_NODE_P (compute_node_count == 1)
#define ZERO_COMPUTE_NODE_P (compute_node_count == 0)
```

These are a number of predicates that allow you to test the identity of the node process in your UDF, using the compute node ID. A compute node’s ID is stored as the global integer variable _myid_ (see Section 9.5). Each of the macros listed below tests certain conditions of _myid_ for a process. For example, the predicate _I_AM_NODE_ZERO_P_ compares the value of _myid_ with the compute node-0 ID and returns TRUE when they are the same. _I_AM_NODESAME_P(n)_ , on the other hand, compares the compute node ID that is passed in _n_ with _myid_. When the two IDs are the same, the function returns TRUE. Node ID predicates are often used in conditional-if statements in UDFs.

```c
/* predicate definitions from para.h header file */

#define I_AM_NODE_HOST_P (myid == node_host)
#define I_AM_NODE_ZERO_P (myid == node_zero)
#define I_AM_NODE_ONE_P (myid == node_one)
#define I_AM_NODE_LAST_P (myid == node_last)
#define I_AM_NODESAME_P(n) (myid == (n))
#define I_AM_NODELESS_P(n) (myid < (n))
#define I_AM_NODEMORE_P(n) (myid > (n))
```

Recall that from Section 9.2, a face may appear in one or two partitions but in order that summation operations don’t count it twice, it is officially allocated to only one of the partitions. The tests above are used with the neighboring cell’s partition ID to
determine if it belongs to the current partition. The convention that is used is that the smaller-numbered compute node is assigned as the “principal” compute node for that face. `PRINCIPAL_FACE_P` returns `TRUE` if the face is located on its principal compute node. The macro can be used as a test condition when you want to perform a global sum on faces and some of the faces are partition boundary faces. (The macro returns `TRUE` for the serial process). Below is the definition of `PRINCIPAL_FACE_P` from `para.h`. See Section 9.2 for more information about `PRINCIPAL_FACE_P`.

```c
/* predicate definitions from para.h header file */
#define PRINCIPAL_FACE_P(f,t) (!TWO_CELL_FACE_P(f,t) || \  
   PRINCIPAL_TWO_CELL_FACE_P(f,t))

#define PRINCIPAL_TWO_CELL_FACE_P(f,t) \  
   (!(I_AM_NODE_MORE_P(C_PART(F_C0(f,t),THREAD_F0(f,t))) || \  
      I_AM_NODE_MORE_P(C_PART(F_C1(f,t),THREAD_F1(f,t)))))
```

### 9.4.4 Global Reduction Macros

Global reduction operations are those that collect data from all of the compute nodes, and reduce the data to a single value, or an array of values. These include operations such as global summations, global maximums and minimums, and global logicals. These macros begin with the prefix `PRF_G` and are defined in `prf.h`. Global summation macros are identified by the suffix `SUM`, global maximums by `HIGH`, and global minimums by `LOW`. The suffixes `AND` and `OR` identify global logicals.

The variable data types for each macro are identified in the macro name, where `R` denotes real data types, `I` denotes integers, and `L` denotes logicals. For example, the macro `PRF_GISUM` finds the summation of integers over the compute nodes.

Each of the global reduction macros discussed in the following sections has two different versions: one takes a single variable argument, while the other takes a variable array. Macros with a 1 appended to the end of the name take one argument, and return a single variable as the global reduction result. For example, the macro `PRF_GIHIGH1(x)` expands to a function that takes one argument `x` and computes the maximum of the variable `x` amongst all of the compute nodes, and returns it. The result can then be assigned to another variable (e.g., `y`) as shown below.

**Example: Global Reduction Variable Macro**

```c
{
   int y;
   int x = myid;
   y = PRF_GIHIGH1(x); /* y now contains the same number (compute_node_count - 1) on all the nodes */
}
```
Macros without a 1 suffix, on the other hand, compute global reduction variable arrays. These macros take three arguments: \texttt{x, N, and iwork} where \texttt{x} is an array, \texttt{N} is the number of elements in the array, and \texttt{iwork} is an array that is of the same type and size as \texttt{x} which is needed for temporary storage. Macros of this type are passed an array \texttt{x} and the elements of array \texttt{x} are filled with the new result after returning from the function. For example, the macro \texttt{PRF.GIHIGH(x,N,iwork)} expands to a function that computes the maximum of each element of the array \texttt{x} over all the compute nodes, uses the array \texttt{iwork} for temporary storage, and modifies array \texttt{x} by replacing each element with its resulting global maximum. The function does not return a value.

**Example: Global Reduction Variable Array Macro**

```c
{
  real x[N], iwork[N];

  /* The elements of \texttt{x} are set in the working array here and will have different values on each compute node. 
   * In this case, \texttt{x[0]} could be the maximum cell temperature of all the cells on the compute node. \texttt{x[1]} the maximum pressure, \texttt{x[2]} the maximum density, etc. */

  PRF_GIHIGH(x,N,iwork); /* The maximum value for each value over all the compute nodes is found here */

  /* The elements of \texttt{x} on each compute node now hold the same maximum values over all the compute nodes for temperature, pressure, density, etc. */
}
```
Global Summations

Macros that can be used to compute global sums of variables are identified by the suffix SUM. PRF\_GISUM1 and PRF\_GISUM compute the global sum of integer variables and integer variable arrays, respectively.

PRF\_GRSUM1(x) computes the global sum of a real variable x across all compute nodes. The global sum is of type float when running a single precision version of FLUENT and type double when running the double precision version. Alternatively, PRF\_GRSUM(x,N,iwork) computes the global sum of a float variable array for single precision and double when running double precision.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRF_GISUM1(x)</td>
<td>Returns sum of integer x over all compute nodes.</td>
</tr>
<tr>
<td>PRF_GISUM(x,N,iwork)</td>
<td>Sets x to contain sums over all compute nodes.</td>
</tr>
<tr>
<td>PRF_GRSUM1(x)</td>
<td>Returns sum of x over all compute nodes; float if single precision, double if double precision.</td>
</tr>
<tr>
<td>PRF_GRSUM(x,N,iwork)</td>
<td>Sets x to contain sums over all compute nodes; float array if single precision, double array if double precision.</td>
</tr>
</tbody>
</table>
### Global Maximums and Minimums

Macros that can be used to compute global maximums and minimums of variables are identified by the suffixes `HIGH` and `LOW`, respectively. `PRF_GIHIGH1` and `PRF_GIHIGH` compute the global maximum of `integer` variables and `integer` variable arrays, respectively.

`PRF_GRIHIGH1(x)` computes the global maximum of a `real` variable `x` across all compute nodes. The value of the global maximum is of type `float` when running the single precision version of FLUENT and type `double` when running the double precision version.

`PRF_GRIHIGH(x,N,iwork)` computes the global maximum of a `real` variable array, similar to the description of `PRF_GRSUM(x,N,iwork)` on the previous page. The same naming convention used for `PRF_GHIGH` macros applies to `PRF_GLOW`.

#### Global Maximums

<table>
<thead>
<tr>
<th>Macro</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PRF_GIHIGH1(x)</code></td>
<td>Returns maximum of integer <code>x</code> over all compute nodes.</td>
</tr>
<tr>
<td><code>PRF_GIHIGH(x,N,iwork)</code></td>
<td>Sets <code>x</code> to contain maximums over all compute nodes.</td>
</tr>
<tr>
<td><code>PRF_GRIHIGH1(x)</code></td>
<td>Returns maximums of <code>x</code> over all compute nodes; <code>float</code> if single precision, <code>double</code> if double precision.</td>
</tr>
<tr>
<td><code>PRF_GRIHIGH(x,N,iwork)</code></td>
<td>Sets <code>x</code> to contain maximums over all compute nodes; <code>float</code> array if single precision, <code>double</code> array if double precision.</td>
</tr>
</tbody>
</table>

#### Global Minimums

<table>
<thead>
<tr>
<th>Macro</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PRF_GILOW1(x)</code></td>
<td>Returns minimum of integer <code>x</code> over all compute nodes.</td>
</tr>
<tr>
<td><code>PRF_GILOW(x,N,iwork)</code></td>
<td>Sets <code>x</code> to contain minimums over all compute nodes.</td>
</tr>
<tr>
<td><code>PRF_GRILOW1(x)</code></td>
<td>Returns minimum of <code>x</code> over all compute nodes; <code>float</code> if single precision, <code>double</code> if double precision.</td>
</tr>
<tr>
<td><code>PRF_GRILOW(x,N,iwork)</code></td>
<td>Sets <code>x</code> to contain minimums over all compute nodes; <code>float</code> array if single precision, <code>double</code> array if double precision.</td>
</tr>
</tbody>
</table>
Global Logicals

Macros that can be used to compute global logical ANDs and logical ORs are identified by the suffixes AND and OR, respectively. PRF.GLOR1(x) computes the global logical OR of variable x across all compute nodes. PRF.GLOR(x,N,iwork) computes the global logical OR of variable array x. The elements of x are set to TRUE if any of the corresponding elements on the compute nodes are TRUE.

By contrast, PRF.GLAND(x) computes the global logical AND across all compute nodes and PRF.GLAND(x,N,iwork) computes the global logical AND of variable array x. The elements of x are set to TRUE if any of the corresponding elements on the compute nodes are TRUE.

Global Logicals

<table>
<thead>
<tr>
<th>Macro</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRF.GLOR1(x)</td>
<td>TRUE when variable x is TRUE for any of the compute nodes</td>
</tr>
<tr>
<td>PRF.GLOR(x,N,iwork)</td>
<td>TRUE when any of the elements in variable array x is TRUE</td>
</tr>
<tr>
<td>PRF.GLAND1(x)</td>
<td>TRUE when variable x is TRUE for all compute nodes</td>
</tr>
<tr>
<td>PRF.GLAND(x,N,iwork)</td>
<td>TRUE when every element in variable array x is TRUE</td>
</tr>
</tbody>
</table>

Global Synchronization

PRF.GSYNC() can be used when you want to globally synchronize compute nodes before proceeding with the next operation. When you insert a PRF.GSYNC macro in your UDF, no commands beyond it will execute until the preceding commands in the source code have been completed on all of the compute nodes. Synchronization may be useful when debugging your function.

9.4.5 Looping Macros

There are three types of cell looping macros that are available for parallel coding; one that loops over interior cells only, exterior cells only, and both interior and exterior cells.

Looping Over Cells

A partitioned grid in parallel FLUENT is made up of interior cells and exterior cells (see Figure 9.2.1). There is a set of cell-looping macros you can use to loop over interior cells only, exterior cells only, or both interior and exterior cells.
9.4 Macros for Parallel UDFs

Figure 9.4.1: Looping Over Interior Cells in a Partitioned Grid Using begin,end_c_loop_int (indicated by the green cells)

Interior Cell Looping Macro

The macro begin,end_c_loop_int loops over interior cells in a partitioned grid (Figure 9.4.1) and is identified by the suffix int. This macro pair can also be used by the serial version of FLUENT to loop over all cells in the given thread. It contains a begin and end statement, and between these statements, operations can be performed on each of the thread's interior cells in turn. The macro is passed a cell index c and a cell thread pointer tc.

```c
begin_c_loop_int(c, tc)
{
    /* C_VOLUME gets the cell volume and accumulates it. The end result will be the total volume of each compute node's respective grid */
    total_volume += C_VOLUME(c,tc);
}
end_c_loop_int(c, tc)
```

Example

```c
real total_volume = 0.0;
begin_c_loop_int(c,tc)
{
    /* C_VOLUME gets the cell volume and accumulates it. The end result will be the total volume of each compute node's respective grid */
    total_volume += C_VOLUME(c,tc);
}
end_c_loop_int(c,tc)
```
Parallel UDF Usage

Figure 9.4.2: Looping Over Exterior Cells in a Partitioned Grid Using begin,end_c_loop_ext (indicated by the green cells)

Exterior Cell Looping Macro

The macro begin,end_c_loop_ext loops over exterior cells in a partitioned grid (Figure 9.4.2) and is identified by the suffix ext. It contains a begin and end statement, and between these statements, operations can be performed on each of the thread’s exterior cells in turn. The macro is passed a cell index c and cell thread pointer tc. In most situations, there is no need to use the exterior cell loop macros. They are only provided for convenience if you come across a special need in your UDF.

begin_c_loop_ext(c, tc)
{
}
end_c_loop_ext(c,tc)
Interior and Exterior Cell Looping Macro

The macro `begin_c_loop` can be used in a serial or parallel UDF. In parallel, the macro will loop over all interior and exterior cells in a grid partition (Figure 9.4.3). Note that in serial, this pair of macros is equivalent to the `begin_c_loop` macros. It contains a `begin` and `end` statement, and between these statements, operations can be performed on each of the thread’s interior and exterior cells in turn. The macro is passed a cell index `c` and a cell thread pointer `tc`.

```c
begin_c_loop(c, tc)
{
}
end_c_loop(c, tc)
```
Parallel UDF Usage

Example

real temp;
beginc_loop(c,tc)
{
    /* get cell temperature, compute temperature function and store
     * result in user-defined memory, location index 0. */
    temp = C_T(c,tc);
    C_UDMI(c,tc,0) = (temp - tmin) / (tmax - tmin);
    /* assumes a valid tmax and tmin has already been computed */
}
end_c_loop(c,tc)

Looping over Faces

For the purpose of discussing parallel FLUENT, faces can be categorized into two types:
interior faces and boundary zone faces (Figure 9.2.2). Partition boundary faces are
interior faces that lie on the partition boundary of a compute node's grid.

There is only one face-looping macro available in parallel FLUENT that loops over
all interior and boundary zone faces in a compute node (Figure 9.4.4). The macro
begin,end_f_loop contains a begin and end statement, and between these statements,
operations can be performed on each of the faces of the thread. The macro is passed a
face index f and face thread pointer tf.

begin_f_loop(f, tf)
{
    
}
end_f_loop(f,tf)

! Face looping macros do not have _int and _ext forms.

Recall that partition boundary faces lie on the boundary between two adjacent compute
nodes and are represented on both nodes. Therefore, there are some computations (e.g.,
summations) when a partition boundary face will get counted twice in a face loop. This
can be corrected by testing whether the current node is a face’s principal compute node
inside your face looping macro, using PRINCIPAL_FACE_P. This is shown in the example
below. See Section 9.2 for details.
9.4 Macros for Parallel UDFs

Figure 9.4.4: Looping Over Faces in a Partitioned Grid Using `begin,end_f_loop`

Example

begin_f_loop(f,tf)
/* each compute node checks whether or not it is the principal compute node with respect to the given face and thread */

if PRINCIPAL_FACE_P(f,tf)
/* face is on the principal compute node, so get the area and pressure vectors, and compute the total area and pressure for the thread from the magnitudes */
{
    F_AREA(area,f,tf);
    total_area += NV_MAG(area);
    total_pres_a += NV_MAG(area)*F_P(f,tf);
}
end_f_loop(f,tf)

total_area = PRF_GRSUM1(total_area);
total_pres_a = PRF_GRSUM1(total_pres_a);
9.4.6 Cell and Face Partition ID Macros

In general, cells and faces have a partition ID that is numbered from 0 to \( n-1 \), where \( n \) is the number of compute nodes. The partition IDs of cells and faces are stored in the variables \( \text{C\_PART} \) and \( \text{F\_PART} \), respectively. \( \text{C\_PART}(c,tc) \) stores the integer partition ID of a cell and \( \text{F\_PART}(f,tf) \) stores the integer partition ID of a face.

Note that \( \text{myid} \) can be used in conjunction with the partition ID, since the partition ID of an exterior cell is the ID of the neighboring compute node.

Cell Partition IDs

For interior cells, the partition ID is the same as the compute node ID. For exterior cells, the compute node ID and the partition ID are different. For example, in a parallel system with two compute nodes (0 and 1), the exterior cells of compute node-0 have a partition ID of 1, and the exterior cells of compute node-1 have a partition ID of 0 (Figure 9.4.5).
Face Partition IDs

For interior faces and boundary zone faces, the partition ID is the same as the compute node ID. The partition ID of a partition boundary face, however, can be either the same as the compute node, or it can be the ID of the adjacent node, depending on what values \( \text{F\_PART} \) is filled with (Figure 9.4.5). Recall that an exterior cell of a compute node has only partition boundary faces; the other faces of the cell belong to the adjacent compute node. Therefore, depending on the computation you want to do with your UDF, you may want to fill the partition boundary face with the same partition ID as the compute node (using \text{Fill\_Face\_Part\_With\_Same}) or with different IDs (using \text{Fill\_Face\_Part\_With\_Different}). Face partition IDs will need to be filled before you can access them with the \text{F\_PART} macro. There is rarely a need for face partition IDs in parallel UDFs.

9.4.7 Message Displaying Macros

You can direct \textsc{FLUENT} to display messages on a host, node, or serial process using the \texttt{Message} utility. To do this, simply use a conditional \texttt{if} statement and the appropriate compiler directive (e.g., \#if RP\_NODE) to select the process(es) you want the message to come from. This is demonstrated in the following example:

Example

\begin{verbatim}
#if RP_NODE
   Message("Total Area Before Summing %f\n",total\_area);
#endif /* RP_NODE */
\end{verbatim}

In this example, the message will be sent by the compute nodes. (It will not be sent by the host or serial process.)

\texttt{Message0} is a specialized form of the \texttt{Message} utility. \texttt{Message0} will send messages from compute node-0 only and is ignored on the other compute nodes, without having to use a compiler directive. Note that \texttt{Message0} will also display messages on a serial process.

Example

\begin{verbatim}
/* Let Compute Node-0 display messages */
Message0("Total volume = %f\n",total\_volume);
\end{verbatim}
9.4.8 Message Passing Macros

High-level communication macros of the form node\_to\_host... and host\_to\_node... that are described in Section 9.4.2 are typically used when you want to send data from the host to all of the compute nodes, or from node-0 to the host. You cannot, however, use these high-level macros when you need to pass data between compute nodes, or pass data from all of the compute nodes to compute node-0. In these cases, you can use special message passing macros described in this section.

Note that the higher-level communication macros expand to functions that perform a number of lower-level message passing operations which send sections of data as single arrays from one process to another process. These lower-level message passing macros can be easily identified in the macro name by the characters SEND and RECV. Macros that are used to send data to processes have the prefix PRF\_CSEND, whereas macros that are used to receive data from processes have the prefix PRF\_CRECV. Data that is to be sent or received can belong to the following data types: character (CHAR), integer (INT), REAL and logical (BOOLEAN). BOOLEAN variables are TRUE or FALSE. REAL variables are assigned as float data types when running a single precision version of FLUENT and double when running double precision. Message passing macros are defined in the prf.h header file and are listed below.

```c
/* message passing macros */
PRF_CSEND_CHAR(to, buffer, nelem, tag)
PRF_CRECV_CHAR(from, buffer, nelem, tag)
PRF_CSEND_INT(to, buffer, nelem, tag)
PRF_CRECV_INT(from, buffer, nelem, tag)
PRF_CSEND_REAL(to, buffer, nelem, tag)
PRF_CRECV_REAL(from, buffer, nelem, tag)
PRF_CSEND_BOOLEAN(to, buffer, nelem, tag)
PRF_CRECV_BOOLEAN(from, buffer, nelem, tag)
```

There are four arguments to the message passing macros. For ‘send’ messages, the argument to is the node ID of the process that data is being sent to. buffer is the name of an array of the appropriate type that will be sent. nelem is the number of elements in the array and tag is a user-defined message tag. The tag convention is to use myid when sending messages and to use the compute node ID of the sender when receiving messages.

For ‘receive’ messages, the argument from is the ID of the sending node. buffer is the name of an array of the appropriate type that will be received. nelem is the number of elements in the array and tag is the ID of the receiving node. The tag convention for receive messages is the ‘from’ node (same as the first argument).
Note that if variables that are to be sent or received are defined in your function as `real` variables, then you can use the message passing macros with the `_REAL` suffix. The compiler will then substitute `PRF_CSENT_DOUBLE` or `PRF_CRECV_DOUBLE` if you are running double precision and `PRF_CSENT_FLOAT` or `PRF_CRECV_FLOAT`, for single precision.

Because message-passing macros are low-level macros, you will need to make sure that when a message is sent from a node process, a corresponding receiving macro appears in the receiving-node process. Note that your UDF cannot directly send messages from a compute node (other than 0) to the host using message-passing macros. They can send messages indirectly to the host through compute node-0. For example, if you want your parallel UDF to send data from all of the compute nodes to the host for postprocessing purposes, the data will first have to be passed from each compute node to compute node-0, and then from compute node-0 to the host. In the case where the compute node processes send a message to compute node-0, compute node-0 must have a loop to receive the \( N \) messages from the \( N \) nodes.

Below is an example of a compiled parallel UDF that utilizes message passing macros `PRF_CSEND` and `PRF_CRECV`. Refer to the comments (*/* in the code, for details about the function.

**Example: Message Passing**

```c
#include "udf.h"
#define WALLID 3

DEFINE_ON_DEMAND(face_p_list)
{
    #if !RP_HOST /* Host will do nothing in this udf. Serial will */
        face_t f;
        Thread *tf;
        Domain *domain;
        real *p_array;
        real x[ND_ND], (*x_array)[ND_ND];
        int n_faces, i, j;

        domain=Get_Domain(1); /* Each Node will be able to access 
                                its part of the domain */

        tf=Lookup_Thread(domain, WALLID); /* Get the thread from the domain */

        /* The number of faces of the thread on nodes 1,2... needs to be sent 
           to compute node-0 so it knows the size of the arrays to receive 
           from each */

        n_faces=THREAD_N_ELEMENTS_INT(tf);
```
/* No need to check for Principal Faces as this UDF
will be used for boundary zones only */

#if RP_NODE
if(! I_AM_NODE_ZERO_P) /* Nodes 1,2... send the number of faces */
{
    PRF_CSEND_INT(node_zero, &n_faces, 1, myid);
}
#endif

/* Allocating memory for arrays on each node */
p_array=(real *)malloc(n_faces*sizeof(real));
x_array=(real (*)[ND_ND])malloc(ND_ND*n_faces*sizeof(real));

begin_f_loop(f, tf)
    /* Loop over interior faces in the thread, filling p_array
with face pressure and x_array with centroid */
    {
        p_array[f] = F_P(f, tf);
        F_CENTROID(x_array[f], f, tf);
    }
end_f_loop(f, tf)

/* Send data from node 1,2, ... to node 0 */
Message0("\n\nstart\n");
#if RP_NODE
if(! I_AM_NODE_ZERO_P) /* Only SEND data from nodes 1,2... */
{
    PRF_CSEND_REAL(node_zero, p_array, n_faces, myid);
    PRF_CSEND_REAL(node_zero, x_array[0], ND_ND*n_faces, myid);
}
else
#endif
{/*
Node-0 and Serial processes have there own data,
so list it out first */
Message0("\n\nList of Pressures...
");
/* Same as Message() on SERIAL */

for(j=0; j<n_faces; j++)
/* n_faces is currently node-0/serial value */
{
    # if RP_3D
        Message0("%12.4e %12.4e %12.4e %12.4e\n",}
x_array[j][0], x_array[j][1], x_array[j][2], p_array[j]);
# else /* 2D */
    Message0("%12.4e %12.4e %12.4e\n",
        x_array[j][0], x_array[j][1], p_array[j]);
# endif
}
}

/* Node-0 must now RECV data from the other nodes and list that too */
#if RP_NODE
    if (I_AM_NODE_ZERO_P)
    {
        compute_node_loop_not_zero(i)
        /* See para.h for definition of this loop */
        {
            PRF_CRECV_INT(i, &n_faces, 1, i);
            /* n_faces now value for node-i */
            /* Reallocate memory for arrays for node-i */
            p_array=(real *)realloc(p_array, n_faces*sizeof(real));
            x_array=(real(*)[ND_ND])realloc(x_array,ND_ND*n_faces*sizeof(real));

            /* Receive data */
            PRF_CRECV_REAL(1, p_array, n_faces, 1);
            PRF_CRECV_REAL(1, x_array[0], ND_ND*n_faces, 1);
            for(j=0; j<n_faces; j++)
            {
                # if RP_3D
                    Message0("%12.4e %12.4e %12.4e %12.4e\n",
                        x_array[j][0], x_array[j][1], x_array[j][2], p_array[j]);
                # else /* 2D */
                    Message0("%12.4e %12.4e %12.4e\n",
                        x_array[j][0], x_array[j][1], p_array[j]);
                # endif
            }
        }
    }
#endif /* RP_NODE */

free(p_array); /* Each array has to be freed before function exit */
free(x_array);

#endif /* ! RP_HOST */
}
9.4.9 Macros for Exchanging Data Between Compute Nodes

EXCHANGE_SVAR_MESSAGE and EXCHANGE_SVAR_FACE_MESSAGE can be used to exchange storage variables (SV...) between compute nodes. EXCHANGE_SVAR_MESSAGE exchanges cell data between compute nodes, while EXCHANGE_SVAR_FACE_MESSAGE exchanges face data. Note that compute nodes are ‘virtually’ synchronized when an EXCHANGE macro is used; receiving compute nodes wait for data to be sent, before continuing.

/* Compute Node Exchange Macros */

EXCHANGE_SVAR_FACE_MESSAGE(domain, (SV_P, SV_NULL));
EXCHANGE_SVAR_MESSAGE(domain, (SV_P, SV_NULL));

EXCHANGE_SVAR_FACE_MESSAGE() is rarely needed in UDFs. You can exchange multiple storage variables between compute nodes. Storage variable names are separated by commas in the argument list and the list is ended by SV_NULL. For example, EXCHANGE_SVAR_MESSAGE(domain, (SV_P, SV_T, SV_NULL)) is used to exchange cell pressure and temperature variables. You can determine a storage variable name from the header file that contains the variable’s definition statement. For example, suppose you want to exchange the cell pressure (C_P) with an adjacent compute node. You can look at the header file that contains the definition of C_P (mem.h) and determine that the storage variable for cell pressure is SV_P. You will need to pass the storage variable to the exchange macro.
9.5 Process Identification

Each process in parallel FLUENT has a unique integer identifier that is stored as the global variable \texttt{myid}. When you use \texttt{myid} in your parallel UDF, it will return the integer ID of the current compute node (including the host). The host process has an ID of \texttt{node_host(=999999)} and is stored as the global variable \texttt{node_host}. Compute node-0 has an ID of 0 and is assigned to the global variable \texttt{node_zero}. Below is a list of global variables in parallel FLUENT.

Global Variables in Parallel FLUENT

\begin{verbatim}
int node_zero = 0;
int node_host = 999999;
int node_one = 1;
int node_serial = 1000000;

int node_last; /* returns the id of the last compute node */
int compute_node_count; /* returns the number of compute nodes */
int myid; /* returns the id of the current compute node (and host) */
\end{verbatim}

\texttt{myid} is commonly used in conditional-if statements in parallel UDF code. Below is some sample code that uses the global variable \texttt{myid}. In this example, the total number of faces in a face thread is first computed by accumulation. Then, if \texttt{myid} is not compute node-0, the number of faces is passed from all of the compute nodes to compute node-0 using the message passing macro \texttt{PRF_CSEND_INT}. (See Section 9.4.8 for details on \texttt{PRF_CSEND_INT}).

Example: Usage of \texttt{myid}

\begin{verbatim}
int noface=0;
begin_f_loop(f, tf) /* loops over faces in a face thread and computes number of faces */
{
  noface++;
}
end_f_loop(f, tf)

/* Pass the number of faces from node 1,2, ... to node 0 */

#if RP_NODE
if(myid!=node_zero)
{
  PRF_CSEND_INT(node_zero, &noface, 1, myid);
}
#endif
\end{verbatim}
9.6 Parallel UDF Example

The following is an example of a serial UDF that has been parallelized, so that it can run on any version of FLUENT (host, node, serial). Explanations for the various changes from the simple serial version are provided in the /* comments */ and discussed below. The UDF, named face_av, is defined using an adjust function, computes a global sum of pressure on a specific face zone, and computes its area average.

Example: Global Summation of Pressure on a Face Zone and its Area Average Computation

#include "udf.h"

DEFINE_ADJUST(face_av,domain)
{
  /* Variables used by serial, host, node versions */
  int surface_thread_id=0;
  real total_area=0.0;
  real total_force=0.0;

  /* "Parallelized" Sections */
  #if !RP_HOST /* Compile this section for computing processes only (serial and node) since these variables are not available on the host */
  Thread* thread;
  face_t face;
  real area[ND_ND];
  #endif /* !RP_HOST */

  /* Get the value of the thread ID from a user-defined Scheme variable */
  #if !RP_NODE /* SERIAL or HOST */
  surface_thread_id = RP_Get_Integer("pres_av/thread-id");
  Message("\nCalculating on Thread # %d\n",surface_thread_id);
  #endif /* !RP_NODE */

  /* To set up this user Scheme variable in cortex type */
  /* (rp-var-define 'pres_av/thread-id 2 'integer #f) */
  /* Once set up you can change it to another thread’s ID using : */
  /* (rpsetvar 'pres_av/thread-id 7) */

  /* Send the ID value to all the nodes */
  host_to_node_int_1(surface_thread_id); /* Does nothing in serial */
#if RP_NODE
    Message("\nNode %d is calculating on thread # %d\n", myid, surface_thread_id);
#endif /* RP_NODE */

#if !RP_HOST /* SERIAL or NODE */
    /* thread is only used on compute processes */
    thread = Lookup_Thread(domain, surface_thread_id);

    begin_f_loop(face, thread)

        /* If this is the node to which face "officially" belongs, */
        /* get the area vector and pressure and increment */
        /* the total area and total force values for this node */
        if (PRINCIPAL_FACE_P(face, thread)) /* Always TRUE in serial version */ {
            F_AREA(area, face, thread);
            total_area += NV_MAG(area);
            total_force += NV_MAG(area)*F_P(face, thread);
        }
    end_f_loop(face, thread)

    Message("Total Area Before Summing %f\n", total_area);
    Message("Total Normal Force Before Summing %f\n", total_force);

    # if RP_NODE /* Perform node synchronized actions here */
        total_area = PRF_GRSUM1(total_area);
        total_force = PRF_GRSUM1(total_force);
    # endif /* RP_NODE */
#endif /* !RP_HOST */

    /* Pass the node’s total area and pressure to the Host for averaging */
    node_to_host_real_2(total_area, total_force); /* Does nothing in SERIAL */
#endif /* !RP_HOST */

    /* Pass the node’s total area and pressure to the Host for averaging */
    node_to_host_real_2(total_area, total_force); /* Does nothing in SERIAL */

#if !RP_NODE /* SERIAL or HOST */
    Message("Total Area After Summing: %f (m2)\n", total_area);
    Message("Total Normal Force After Summing %f (N)\n", total_force);
    Message("Average pressure on Surface %d is %f (Pa)\n", 
        surface_thread_id, (total_force/total_area));
#endif /* !RP_NODE */
Parallel UDF Usage

The function begins by initializing the variables `surface_thread_id`, `total_area`, and `total_force` for all processes. This is done because the variables are used by the serial, host, and node processes. The compute nodes use the variables for computation purposes and the host uses them for message-passing and displaying purposes. Next, the preprocessor is directed to compile `thread`, `face`, and `area` variables only on the serial and node versions (and not the host), since faces and threads are only defined in the serial and node versions of FLUENT. (Note that in general, the host will ignore these statements since its face and cell data are zero, but it is good programming practice to exclude the host. See Section 9.4 for details on compiler directives.)

Next, a user-defined Scheme variable named `pres_av/thread-id` is obtained by the host (and serial) process using the `RP_Get_Integer` utility (see Section 6.8), and is assigned to the variable `surface_thread_id`. (Note that this user-defined Scheme variable was previously set up in Cortex and assigned a value of 2 by typing the text commands shown in the comments.) Once a Scheme-based variable is set up for the thread ID, it can be easily changed to another thread ID from the text interface, without the burden of modifying the source code and recompiling the UDF. Since the host communicates with Cortex and the nodes are not aware of Scheme variables, it is essential to direct the compiler to exclude the nodes from compiling them using `#if !RP_NODE`. Failing to bracket the Scheme variable statements within the host will result in a compile error.

The `surface_thread_id` is then passed from the host to compute node-0 using the `host_to_node` macro. Compute node-0, in turn, automatically distributes the variable to the other compute nodes. The serial and node processes are directed to loop over all faces in the thread associated with the `surface_thread_id`, using `#if !RP_HOST`, and compute the total area and total force. Since the host does not contain any thread data, it will ignore these statements if you do not direct the compiler, but it is good programming practice to do so. The macro `PRINCIPAL_FACE_P` is used to ensure that faces at partition boundaries are not counted twice (see Section 9.2). The nodes display the total area and force on the monitors (using the `Message` utility) before the global summation. `PRF_GRSUM1` (Section 9.4.4) is a global summation macro that is used to compute the total area and force of all the compute nodes. These operations are directed for the compute nodes using `#if RP_NODE`.

9-38 © Fluent Inc. January 22, 2003
9.7 Writing Files in Parallel

Although compute nodes can perform computations on data simultaneously when FLUENT is running in parallel, when data is written to a single, common file, the writing operations have to be sequential. The file has to be opened and written to by processes that have access to the desired file system. It is often the case that the compute nodes are running on a dedicated parallel machine without disk space. This means that all of the data has to be written from the host process which always runs on a machine with access to a file system, since it reads and writes the case and data files. This implies that unlike the example in Section 9.4.8, where data is only passed to compute node-0 to be collated, data must now be passed from all the compute nodes to compute node-0, which then passes it on to the host node which writes it to the file. This process is known as “marshalling”.

Thus, file writing in parallel is done in the following stages:

1. The host process opens the file.
2. Compute node-0 sends its data to the host.
3. The other compute nodes send their data to compute node-0.
4. Compute node-0 receives the data from the other compute nodes and sends it to the host.
5. The host receives the data sent from all the compute nodes and writes it to the file.
6. The host closes the file.

Since the SERIAL, HOST, and NODE processes are performing different tasks, the example below appears long and utilizes a large number of compiler directives. If, however, as an exercise you make three copies of this example and in each copy delete the unused sections for either the SERIAL, HOST or NODE versions, then you will see that it is actually quite a simple routine.

Example: Writing Data to a Common File on the Host Process’s File System

/*****************************/
This function will write pressures and positions
for a fluid zone to a file on the host machine
/*****************************/
#include "udf.h"

#define FLUID_ID 2
DEFINE_ON_DEMAND(pressures_to_file)
{
    /* Different variables are needed on different nodes */
#if !RP_HOST
    Domain *domain=Get_Domain(1);
    Thread *thread;
    cell_t c;
#else
    int i;
#endif

#if !RP_NODE
    FILE *fp = NULL;
    char filename[]="press_out.txt";
#endif

#if PARALLEL
    int size; /* data passing variables */
    real *array;
    int pe;
#endif

/* Only Serial and Compute Nodes have data on threads */
#if !RP_HOST
    thread=Lookup_Thread(domain,FLUID_ID);
#endif

#if !RP_NODE /* SERIAL or HOST */
    if ((fp = fopen(filename, "w"))==NULL)
        Message("\n Warning: Unable to open %s for writing\n",filename);
    else
        Message("\nWriting Pressure to %s...",filename);
#endif

/* UDF Now does 3 different things depending on SERIAL, NODE or HOST */
#if !PARALLEL /* SERIAL */
    begin_c_loop(c,thread)
        fprintf(fp, "%g\n", C_P(c,thread)); /* Simply write out pressure data */
    end_c_loop(c,thread)
#endif /* !PARALLEL */

#if RP_NODE
/* Each Node loads up its data passing array */
size=THREAD_W_ELEMENTS_INT(thread);
array = (real *)malloc(size * sizeof(real));

begin_c_loop_int(c,thread)
  array[c]= C_P(c,thread);
end_c_loop_int(c,thread)

/* Set pe to destination node */
/* If on node_0 send data to host */
/* Else send to node_0 because */
/* compute nodes connect to node_0 & node_0 to host */
pe = (I_AM_NODE_ZERO_P) ? node_host : node_zero;

PRF_CSEND_INT(pe, &size, 1, myid);
PRF_CSEND_REAL(pe, array, size, myid);

free(array); /* free array on nodes once data sent */

/* node_0 now collect data sent by other compute nodes */
/* and sends it straight on to the host */
if (I_AM_NODE_ZERO_P)
  compute_node_loop_not_zero (pe)
  {
    PRF_CRECV_INT(pe, &size, 1, pe);
    array = (real *)malloc(size * sizeof(real));
    PRF_CRECV_REAL(pe, array, size, pe);

    PRF_CSEND_INT(node_host, &size, 1, myid);
    PRF_CSEND_REAL(node_host, array, size, myid);

    free((char *)array);
  }
#endif /* RP_NODE */

#if RP_HOST
compute_node_loop (pe) /* only acts as a counter in this loop */
  {
    /* Receive data sent by each node and write it out to the file */
    PRF_CRECV_INT(node_zero, &size, 1, node_zero);
    array = (real *)malloc(size * sizeof(real));
    PRF_CRECV_REAL(node_zero, array, size, node_zero);
for (i=0; i<size; i++)
    fprintf(fp, "%g\n", array[i]);

free(array);
}
#endif /* RP_HOST */

#if !RP_NODE /* SERIAL or HOST */
    fclose(fp); /* Close the file that was only opened if on SERIAL or HOST */
    Message("Done\n");
#endif

}
Chapter 10. User-Defined Scalar (UDS) Transport Modeling

This chapter provides an overview of user-defined scalars and their usage in FLUENT.

- Section 10.1: Introduction
- Section 10.2: Theory
- Section 10.3: Defining, Solving, and Postprocessing a UDS

10.1 Introduction

FLUENT can solve the transport equation for an arbitrary, user-defined scalar (UDS) in the same way that it solves the transport equation for a scalar such as species mass fraction. Extra scalar transport equations may be needed in certain types of combustion applications or in plasma-enhanced surface reaction modeling, for example. User-defined scalars can be used to implement a magnetohydrodynamic (MHD) model so that flow of a conducting fluid in a magnetic field can be solved. In MHD, the flow of the conducting fluid induces a magnetic field, which is solved using user-defined scalars. The magnetic field creates a resistance to the flow, which is modeled using user-defined source terms. See Sections 4.3.19 and 4.3.20 for examples of writing UDFs to customize scalar transport equations.

10.2 Theory

Single Phase Flow

For an arbitrary scalar $\phi_k$, FLUENT solves the equation

$$\frac{\partial \rho \phi_k}{\partial t} + \frac{\partial}{\partial x_i}(\rho u_i \phi_k - \Gamma_k \frac{\partial \phi_k}{\partial x_i}) = S_{\phi_k} \quad k = 1, ..., N$$

(10.2-1)

where $\Gamma_k$ and $S_{\phi_k}$ are the diffusion coefficient and source term supplied by you for each of the $N$ scalar equations. For the steady-state case, FLUENT will solve one of the three following equations, depending on the method used to compute the convective flux:
User-Defined Scalar (UDS) Transport Modeling

- If convective flux is not to be computed, FLUENT will solve the equation

\[-\frac{\partial}{\partial x_i}(\Gamma_k \frac{\partial \phi_k}{\partial x_i}) = S_{\phi_k} \quad k = 1, \ldots, N \tag{10.2-2}\]

where \( \Gamma_k \) and \( S_{\phi_k} \) are the diffusion coefficient and source term supplied by you for each of the \( N \) scalar equations.

- If convective flux is to be computed with mass flow rate, FLUENT will solve the equation

\[\frac{\partial}{\partial x_i}(\rho u_i \phi_k - \Gamma_k \frac{\partial \phi_k}{\partial x_i}) = S_{\phi_k} \quad k = 1, \ldots, N \tag{10.2-3}\]

- It is also possible to specify a user-defined function to be used in the computation of convective flux. In this case, the user-defined mass flux is assumed to be of the form

\[F = \int_S \rho \vec{u} \cdot d\vec{S} \tag{10.2-4}\]

where \( \vec{dS} \) is the face vector area

! In FLUENT, the user-defined scalars are solved only in fluid cells, not in solid cells.

Multiphase Flow

For multiphase flow, FLUENT solves transport equations for two types of scalars: per phase scalar and mixture. For an arbitrary \( k \) scalar in phase-1, denoted by \( \phi^k_1 \), FLUENT solves the transport equation inside the volume occupied by phase-1

\[\frac{\partial \alpha_l \rho_l \phi^k_1}{\partial t} + \nabla \cdot (\alpha_l \rho_l \vec{u}_l \phi^k_1 - \alpha_l \Gamma^k_1 \nabla \phi^k_1) = S^k_l \quad k = 1, \ldots, N \tag{10.2-5}\]

where \( \alpha_l \), \( \rho_l \), and \( \vec{u}_l \) are the volume fraction, physical density, and velocity of phase-1, respectively. \( \Gamma^k_1 \) and \( S^k_l \) are the diffusion coefficient and source term, respectively, which you will need to specify. In this case, scalar \( \phi^k_1 \) is associated only with one phase (phase-1) and is considered an individual field variable of phase-1.

The mass flux for phase-1 is defined as

\[F_1 = \int_S \alpha_l \rho_l \vec{u}_l \cdot d\vec{S} \tag{10.2-6}\]

If the transport variable described by scalar \( \phi^k_1 \) represents the physical field that is shared between phases, or is considered the same for each phase, then you should consider this
10.3 Defining, Solving, and Postprocessing a UDS

scalar as being associated with a mixture of phases, $\phi^k$. In this case, the generic transport equation for the scalar is

$$\frac{d}{dt} \rho_m \phi^k + \nabla \cdot (\rho_m \bar{u}_m \phi^k - \Gamma^k \nabla \phi^k) = S^k \quad k = 1, \ldots, N$$

(10.2-7)

where mixture density $\rho_m$, mixture velocity $\bar{u}_m$, and mixture diffusivity for the scalar $k$ $\Gamma^k$ are calculated according to

$$\rho_m = \sum_l \alpha_l \rho_l$$

(10.2-8)

$$\rho_m \bar{u}_m = \sum_l \alpha_l \rho_l \bar{u}_l$$

(10.2-9)

$$F_m = \int_S \rho \bar{u}_m \cdot d\vec{S}$$

(10.2-10)

$$\Gamma^k = \sum_l \alpha_l \Gamma^k_l$$

(10.2-11)

$$S^k_l = \sum_l S^k$$

(10.2-12)

To calculate mixture diffusivity, you will need to specify individual diffusivities for each material associated with individual phases.

Note that if the user-defined mass flux option is activated, then mass fluxes shown in Equation 10.2-6 and Equation 10.2-10 will need to be replaced in the corresponding scalar transport equations.

10.3 Defining, Solving, and Postprocessing a UDS

The procedure for defining, solving, and postprocessing a user-defined scalar (UDS) is outlined below. Note that a significant difference between a UDS for a single-phase versus a multiphase application is that you will need to associate each UDS with its corresponding phase domain or mixture domain, depending on your application. If you supply UDFs for transient terms, convective fluxes, and sources, you will need to be aware that they are directly called from the phase or mixture domains, according to the scalar association settings. See Section 4.3.19 and 4.3.20 for DEFINE macros that apply to user-defined scalar (UDS) transport equations. See Section 11.5 for some examples of user-defined scalar transport applications.
Single Phase Flow

1. Specify the number of scalars in the **User-Defined Scalars** panel (Figure 10.3.1).

   ![User-Defined Scalars Panel](image)

   - The maximum number of user-defined scalar transport equations you can define is 50.

   Figure 10.3.1: The **User-Defined Scalars** Panel

2. Specify the **Flux Function** to be *none*, *mass flow rate*, or a user-defined function. User-defined scalar flux functions are defined using the `DEFINE_UDS_FLUX` macro (see Section 4.3.19). The names of all user-defined functions that have been defined will appear in the **Flux Function** list. The flux function determines how the convective flux is computed, which determines the equation that **FLUENT** solves for the UDS. Selecting *none*, *mass flow rate*, or a user-defined function results in **FLUENT** solving Equation 10.2-2, 10.2-3, or 10.2-4, respectively.

   ! You will specify a single **Flux Function** for all UDS’s. If you have multiple UDS’s, they will all compute the convective flux in the same way. If you select a user-defined function, it should include the flux functions for all UDS’s.

3. Specify the **Unsteady Function** to be *none*, *default*, or a user-defined function. (The names of all user-defined functions that have been defined will appear in the **Unsteady Function** list.) Select *none* for a steady state solution, select *default* if you want the transient term in Equation 10.2-1 to be solved, or select *user-defined* to enable the UDF you defined using the `DEFINE_UDS_UNSTEADY` macro. See Section 4.3.20 for more details.

4. Specify boundary conditions for the UDS on all wall, inflow, and outflow boundaries. You can define a specific value or a specific flux for each scalar.

   ![Define Boundary Conditions](image)

   - (a) Under **User Defined Scalar Boundary Condition** (see, for example, Figure 10.3.2), select either **Specified Flux** or **Specified Value** in the drop-down list next to each scalar.
Figure 10.3.2: The Velocity Inlet Panel with Inputs for a User-Defined Scalar
(b) Under User Defined Scalar Boundary Value, enter a constant value or select a user-defined function for each scalar. If you selected Specified Flux, your input will be the value of the flux at the boundary (i.e., the negative of the term in parenthesis on the left hand side of Equation 10.2-2 dot [as in the dot product of] \( n \) [as in the vector, \( n \)], where \( n \) is the normal into the domain); if you selected Specified Value, your input will be the value of the scalar itself at the boundary.

5. If you want to include a source term in a UDS equation, enable the Source Terms option in the Fluid panel and set the source term for the scalar as a constant value or as a user-defined function (see Figure 10.3.3). If you specify a user-defined function for the source term, your function is defined using DEFINE_SOURCE and it must calculate the source term, \( S_{\phi_k} \), and its derivative, \( \frac{\partial S_{\phi_k}}{\partial \phi_k} \) (see Section 4.3.15).

6. Set the solution parameters, specify an initial value for each UDS (as you do for all other scalar transport equations), and calculate a solution.

7. Examine the results using the usual postprocessing tools. In each postprocessing
panel, the list of field variables will include the **User Defined Scalars**... category, which contains the value of each UDS and its diffusion coefficient \( (\Gamma_k \text{ in Equations 10.2-1, 10.2-2, 10.2-3, or 10.2-4})\):

- Scalar-\(n\)
- Diffusion Coef. of Scalar-\(n\)

**Multiphase Flow**

1. Specify the number of scalars in the **User-Defined Scalars** panel (Figure 10.3.4).

   The maximum number of user-defined scalar transport equations you can define is 50.

   ![Figure 10.3.4: The User-Defined Scalars Panel](image)

2. Specify the **Flux Function** to be none, mass flow rate, or a user-defined function. User-defined scalar flux functions are defined using the **DEFINE_UDS_FLUX** macro (see Section 4.3.19). The names of all user-defined functions that have been defined will appear in the **Flux Function** list. The flux function determines how the convective flux is computed, which determines the equation that **FLUENT** solves for the UDS. Selecting none, mass flow rate, or a user-defined function results in **FLUENT** solving Equation 10.2-2, 10.2-3, or 10.2-4, respectively.

   You will specify a single **Flux Function** for all UDS’s. If you have multiple UDS’s, they will all compute the convective flux in the same way. If you select a user-defined function, it should include the flux functions for all UDS’s.

3. Specify the **Unsteady Function** to be none, default, or a user-defined function. (The names of all user-defined functions that have been defined will appear in the **Unsteady Function** list.) Select none for a steady state solution, select default if you
want the transient term in Equation 10.2-1 to be solved, or select user-defined to enable the UDF you defined using the DEFINE_UDS_UNSTEADY macro. See Section 4.3.20 for more details.

4. Specify the domain that the transport equations will be solved in for each user-defined scalar by choosing the appropriate phase (e.g., water) or mixture in the drop-down list for each scalar (Scalar-0, Scalar-1, etc.) under Domains to solve in. (Use the scroll bar to view additional scalars.) The default association type is set to mixture for all scalars. Click OK when all the domains are specified (Figure 10.3.4).

5. Specify boundary conditions for the UDS on all wall, inflow, and outflow boundaries. You can define a specific value or a specific flux for each scalar.

(a) In the Boundary Conditions panel, select a boundary Zone (e.g., pressure-inlet-4) and Type and then specify the phase or mixture in the drop-down list under Phases and click Set.... This will open the particular boundary zone panel (e.g., Pressure Inlet panel) and allow you to set the UDS boundary parameters on a per-phase (or mixture) basis (Figure 10.3.6).
(b) Under User Defined Scalar Boundary Condition (Figure 10.3.6), select either Specified Flux or Specified Value in the drop-down list next to each scalar.

(c) Under User Defined Scalar Boundary Value, enter a constant value or select a user-defined function (e.g., udf pressure_profile) for each scalar. If you selected Specified Flux, your input will be the value of the flux at the boundary (i.e., the negative of the term in parenthesis on the left hand side of Equation 10.2-2 dot $[as in the dot product of] \mathbf{n} [as in the vector, \mathbf{n}], where \mathbf{n} is the normal into the domain); if you selected Specified Value, your input will be the value of the scalar itself at the boundary.

Repeat Step 5 to specify the boundary condition and value for each user-defined scalar.
6. If you want to include a source term in a UDS equation, enable the **Source Terms** option in the **Fluid** panel and set the source term for the scalar as a constant value or as a user-defined function (see Figure 10.3.7). If you specify a user-defined function for the source term, your function is defined using **DEFINE_SOURCE** and it must calculate the source term, \( S_{\phi_k} \), and its derivative, \( \frac{\partial S_{\phi_k}}{\partial \phi_k} \) (see Section 4.3.15).

![Fluid Panel with Source Terms](image)

**Figure 10.3.7:** The **Fluid** Panel with Inputs for Source Terms for a User-Defined Scalar

7. Set the solution parameters, specify an initial value for each UDS (as you do for all other scalar transport equations), and calculate a solution.

8. Examine the results using the usual postprocessing tools. In each postprocessing panel, the list of field variables will include the **User Defined Scalars**... category, which contains the value of each UDS and its diffusion coefficient (\( \Gamma_k \) in Equation 10.2-1, 10.2-2, 10.2-3, or 10.2-4):

- **Scalar-n**
- **Diffusion Coef. of Scalar-n**
Chapter 11. Sample Problems

This chapter contains sample applications of UDFs in FLUENT.

- Section 11.1: Boundary Conditions
- Section 11.2: Source Terms
- Section 11.3: Physical Properties
- Section 11.4: Reaction Rates
- Section 11.5: User-Defined Scalars

11.1 Boundary Conditions

This section contains two applications of boundary condition UDFs.

- Parabolic Velocity Inlet Profile for a Turbine Vane
- Transient Velocity Inlet Profile for Flow in a Tube
Parabolic Velocity Inlet Profile in a Turbine Vane

Consider the turbine vane illustrated in Figure 11.1.1. An unstructured grid is used to model the flow field surrounding the vane. The domain extends from a periodic boundary on the bottom to an identical one on the top, a velocity inlet on the left, and a pressure outlet on the right.

![Grid Turbine Vane](image)

Figure 11.1.1: The Grid for the Turbine Vane Example

A flow field in which a constant $x$ velocity is applied at the inlet will be compared with one where a parabolic $x$ velocity profile is applied. While the application of a profile using a piecewise-linear profile is available with the boundary profiles option, the specification of a polynomial can only be accomplished by a user-defined function.

The results of a constant-velocity applied field (of 20 m/sec) at the inlet are shown in Figures 11.1.2 and 11.1.3. The initial constant velocity field is distorted as the flow moves around the turbine vane.

The inlet $x$ velocity will now be described by the following profile:

$$v_x = 20 - 20 \left( \frac{y}{0.0745} \right)^2$$

where the variable $y$ is 0.0 at the center of the inlet, and extends to values of $\pm 0.0745$ m at the top and bottom. Thus the $x$ velocity will be 20 m/sec at the center of the inlet, and 0 at the edges.
Figure 11.1.2: Velocity Magnitude Contours for a Constant Inlet $x$ Velocity

Figure 11.1.3: Velocity Vectors for a Constant Inlet $x$ Velocity
A UDF is used to introduce this parabolic profile at the inlet. The C source code (vprofile.c) is shown below. The function makes use of Fluent-supplied solver functions that are described in Section 5.4.

The UDF, named inlet_x_velocity, is defined using DEFINE_PROFILE and has two arguments: thread and position. Thread is a pointer to the face’s thread, and position is an integer that is a numerical label for the variable being set within each loop.

The function begins by declaring variable f as a face_t data type. A one-dimensional array x and variable y are declared as real data types. A looping macro is then used to loop over each face in the zone to create a profile, or an array of data. Within each loop, F_CENTROID outputs the value of the face centroid (array x) for the face with index f that is on the thread pointed to by thread. The y coordinate stored in x[1] is assigned to variable y, and is then used to calculate the x velocity. This value is then assigned to F_PROFILE, which uses the integer position (passed to it by the solver based on your selection of the UDF as the boundary condition for x velocity in the Velocity Inlet panel) to set the x velocity face value in memory.
To make use of this UDF in FLUENT, you will first need to interpret (or compile) the function, and then hook it to FLUENT using the graphical user interface. Follow the procedure for interpreting source files using the Interpreted UDFs panel (Section 7.2), or compiling source files using the Compiled UDFs panel (Section 7.3).

To hook the UDF to FLUENT as the velocity boundary condition for the zone of choice, open the Velocity Inlet panel.

![Define Boundary Conditions...](image)

In the X-Velocity drop-down list, select `udf.inlet_x.velocity`, the name that was given to the function above. This function will now be used, rather than the value of (in this example) 0 that appears in the X-Velocity field. Click on OK to accept the new boundary condition and close the panel.

After the solution is run to convergence, a revised velocity field is obtained as shown in Figures 11.1.4 and 11.1.5. The velocity field shows a maximum at the center of the inlet, which drops to zero at the edges.
Figure 11.1.4: Velocity Magnitude Contours for a Parabolic Inlet $x$ Velocity

Figure 11.1.5: Velocity Vectors for a Parabolic Inlet $x$ Velocity
Transient Velocity Inlet Profile for Flow in a Tube

In this example, a temporally periodic velocity boundary condition will be applied to the inlet of a tube using a UDF. The velocity has the form

\[ v_x = v_0 + A \sin(\omega t) \]

The tube is 1 m long with a radius of 0.2 m. It is assumed to be filled with air with a density of 1 kg/m\(^3\) and a viscosity of \(2 \times 10^{-5}\) kg/m-s. The velocity of the air fluctuates about an equilibrium value, \(v_0\), of 20 m/s, with an amplitude of 5 m/s and at a frequency of 10 rad/s.

The source file listing for the UDF that describes the transient inlet profile is shown below. The function, named \texttt{unsteady_velocity}, is defined using the \texttt{DEFINE_PROFILE} macro. The utility \texttt{CURRENT\_TIME} is used to look up the \texttt{real} flow time, which is assigned to the variable \(t\). (See Section 6.7 for details on \texttt{CURRENT\_TIME}).

```
/******************************************************************************
  unsteady.c
  UDF for specifying a transient velocity profile boundary condition
******************************************************************************

#include "udf.h"

DEFINE_PROFILE(unsteady_velocity, thread, position)
{
  face_t f;
  real t = CURRENT\_TIME;

  begin_f_loop(f, thread)
  {
    F\_PROFILE(f, thread, position) = 20. + 5.0*sin(10.*t);
  }
  end_f_loop(f, thread)
}
```
Before you can interpret or compile the UDF, you must specify an unsteady flow calculation in the Solver panel. Then, follow the procedure for interpreting source files using the Interpreted UDFs panel (Section 7.2), or compiling source files using the Compiled UDFs panel (Section 7.3).

The sinusoidal velocity boundary condition defined by the UDF can now be hooked to the inlet zone for the X-Velocity. In the Velocity Inlet panel, simply select \texttt{udf unsteady\_velocity} in the drop-down list to the right of the X-Velocity field, and click on OK.

The time-stepping parameters are set in the Iterate panel.

\texttt{Solve} \rightarrow \texttt{Iterate}...
In this example, a Time Step Size of 0.0314 s is used so that 20 time steps will complete a full period of oscillation in the inlet velocity. The UDF Profile Update Interval is set to 1 so that the velocity will be updated every iteration. After 60 time steps (or 3 periods) are complete, you can examine the velocity magnitude across the pressure outlet for its response to the oscillating inlet condition.
Sample Problems

To collect this information during the calculation, open the **Surface Monitors** panel before beginning to iterate.

![Surface Monitors panel](image)

Increase the **Surface Monitors** index to 1. This will enable you to define the parameters of `monitor-1` (which you could rename, if desired, in the text entry box under **Name**). Select **Plot** so that the selected quantity will be plotted as the calculation proceeds. Select **Print** to see the changing values of the selected quantity in the console window. Select **Write** so that the information will be written to a file, which will be given the name `monitor-1.out`. (If you change the name of the monitor, that name will be used as the prefix for the output file.)

Under **Every**, you can choose **Iteration**, **Time Step**, or **Flow Time**. To monitor the result of each time step, you should choose the **Time Step** option. By clicking on **Define...** you can specify the quantity to be monitored in the **Define Surface Monitor** panel.
In this example, Velocity... and Velocity Magnitude are chosen in the drop-down lists under Report Of. The location of the report is pressure-outlet-5, which is selected in the Surfaces list. A simple Area-Weighted Average is chosen in the Report Type drop-down list, with the Flow Time chosen in the X Axis drop-down list.

Once the first time step has been completed, the monitor should appear in the chosen plot window. Alternatively, you can read the file by opening the File XY Plot panel.

You can read the output file by typing its name in the text entry box under Files and clicking on Add.... By selecting this file and clicking on Plot, you can obtain the plot shown in Figure 11.1.6.
Figure 11.1.6: Average Velocity Magnitude at the Pressure Outlet

The figure nicely illustrates that the velocity oscillates around the equilibrium value, 20 m/s, with an amplitude of 5 m/s, as expected.

11.2 Source Terms

This section contains an application of a source term UDF. It is executed as an interpreted UDF in FLUENT.

11.2.1 Adding a Momentum Source to a Duct Flow

When a source term is being modeled with a UDF, it is important to understand the context in which the function is called. When you add a source term, FLUENT will call your function as it performs a global loop on cells. Your function should compute the source term and return it to the solver.

In this example, a momentum source will be added to a 2D Cartesian duct flow. The duct is 4 m long and 2 m wide, and will be modeled with a symmetry boundary through the middle. Liquid metal (with properties listed in Table 11.2.1) enters the duct at the left with a velocity of 1 mm/s at a temperature of 290 K. After the metal has traveled 0.5 m along the duct, it is exposed to a cooling wall, which is held at a constant temperature of 280 K. To simulate the freezing of the metal, a momentum source is applied to the metal as soon as its temperature falls below 288 K. The momentum source is proportional to the $x$ component of the velocity, $v_x$, and has the opposite sign:
\[ S_x = -C v_x \]  

(11.2-1)

where \( C \) is a constant. As the liquid cools, its motion will be reduced to zero, simulating the formation of the solid. (In this simple example, the energy equation will not be customized to account for the latent heat of freezing. The velocity field will be used only as an indicator of the solidification region.)

The solver linearizes source terms in order to enhance the stability and convergence of a solution. To allow the solver to do this, you need to specify the dependent relationship between the source and solution variables in your UDF, in the form of derivatives. The source term, \( S_x \), depends only on the solution variable, \( v_x \). Its derivative with respect to \( v_x \) is

\[ \frac{\partial S_x}{\partial v_x} = -C \]  

(11.2-2)

The following UDF specifies a source term and its derivative. The function, named `cell_x_source`, is defined on a cell using `DEFINE_SOURCE`. The constant \( C \) in Equation 11.2-1 is called `CON` in the function, and it is given a numerical value of 20 kg/m\(^3\)-s, which will result in the desired units of N/m\(^3\) for the source. The temperature at the cell is returned by `CT(cell,thread)`. The function checks to see if the temperature is below (or equal to) 288 K. If it is, the source is computed according to Equation 11.2-1 (`CU` returns the value of the \( x \) velocity of the cell). If it is not, the source is set to 0.0. At the end of the function, the appropriate value for the source is returned to the FLUENT solver.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density</td>
<td>8000 kg/m(^3)</td>
</tr>
<tr>
<td>Viscosity</td>
<td>( 5.5 \times 10^{-3} ) kg/m-s</td>
</tr>
<tr>
<td>Specific Heat</td>
<td>680 J/kg-K</td>
</tr>
<tr>
<td>Thermal Conductivity</td>
<td>30 W/m-K</td>
</tr>
</tbody>
</table>

Table 11.2.1: Properties of the Liquid Metal
UDF that adds momentum source term and derivative to duct flow

#include "udf.h"

#define CON 20.0

DEFINE_SOURCE(cell_x_source, cell, thread, dS, eqn)
{
    real source;

    if (C_T(cell,thread) <= 288.)
    {
        /* source term */
        source = -CON*C_U(cell,thread);

        /* derivative of source term w.r.t. x-velocity. */
        dS[eqn] = -CON;
    }
    else
    {
        source = dS[eqn] = 0.;
    }

    return source;
}

To make use of this UDF in FLUENT, you will first need to interpret (or compile) the function, and then hook it to FLUENT using the graphical user interface. Follow the procedure for interpreting source files using the Interpreted UDFs panel (Section 7.2), or compiling source files using the Compiled UDFs panel (Section 7.3).
To include source terms in the calculation, you will first need to activate Source Terms in the Fluid panel.

You can then add a momentum source term by selecting the udf cell_x_source in the drop-down list for X Momentum and clicking OK.

Once the solution has converged, you can view contours of static temperature to see the cooling effects of the wall on the liquid metal as it moves through the duct (Figure 11.2.1).

Contours of velocity magnitude (Figure 11.2.2) show that the liquid in the cool region near the wall has indeed come to rest to simulate solidification taking place.

The solidification is further illustrated by line contours of stream function (Figure 11.2.3).

To more accurately predict the freezing of a liquid in this manner, an energy source term would be needed, as would a more accurate value for the constant appearing in Equation 11.2-1.
Figure 11.2.1: Temperature Contours Illustrating Liquid Metal Cooling

Figure 11.2.2: Velocity Magnitude Contours Suggesting Solidification
Figure 11.2.3: Stream Function Contours Suggesting Solidification
11.3 Physical Properties

This section contains an application of a physical property UDF. It is executed as an interpreted UDF in FLUENT.

11.3.1 Solidification via a Temperature-Dependent Viscosity

UDFs for properties (as well as sources) are called from within a loop on cells. For this reason, functions that specify properties are only required to compute the property for a single cell, and return the value to the FLUENT solver.

The UDF in this example generates a variable viscosity profile to simulate solidification, and is applied to the same problem that was presented in Section 11.2.1. The viscosity in the warm \( (T > 288 \text{ K}) \) fluid has a molecular value for the liquid \( (5.5 \times 10^{-3} \text{ kg/m-s}) \), while the viscosity for the cooler region \( (T < 286 \text{ K}) \) has a much larger value \( (1.0 \text{ kg/m-s}) \). In the intermediate temperature range \( (286 \text{ K} \leq T \leq 288 \text{ K}) \), the viscosity follows a linear profile (Equation 11.3-1) that extends between the two values given above:

\[
\mu = 143.2135 - 0.49725T \tag{11.3-1}
\]

This model is based on the assumption that as the liquid cools and rapidly becomes more viscous, its velocity will decrease, thereby simulating solidification. Here, no correction is made for the energy field to include the latent heat of freezing. The C source code for the UDF is shown below.

The function, named `cell_viscosity`, is defined on a cell using `DEFINE_PROPERTY`. Two real variables are introduced: `temp`, the value of \( C_T(\text{cell,thread}) \), and `mu_lam`, the laminar viscosity computed by the function. The value of the temperature is checked, and based upon the range into which it falls, the appropriate value of `mu_lam` is computed. At the end of the function, the computed value for `mu_lam` is returned to the solver.

```c
#include "udf.h"

DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
    real mu_lam;
    real temp = C_T(cell, thread);

    if (temp > 288.)
        mu_lam = 5.5e-3;
```
else if (temp > 286.)
    mu_lam = 143.2135 - 0.49725 * temp;
else
    mu_lam = 1.;

return mu_lam;
}

This function can be executed as an interpreted or compiled UDF in FLUENT. Follow the procedure for interpreting source files using the Interpreted UDFs panel (Section 7.2), or compiling source files using the Compiled UDFs panel (Section 7.3).

To make use of the user-defined property in FLUENT, you will use the Materials panel. In the drop-down list for Viscosity, select the user-defined option.

Once you select this option, the User-Defined Functions panel opens, from which you can select the appropriate function name. In this example, only one option is available, but in other examples, you may have several functions from which to choose. (Recall that if
you need to compile more than one interpreted UDF, the functions can be concatenated in a single source file prior to compiling.)

The results of this model are similar to those obtained in Section 11.2.1. Figure 11.3.1 shows the viscosity field resulting from the application of the user-defined function. The viscosity varies rapidly over a narrow spatial band from a constant value of 0.0055 to 1.0 kg/m-s.

The velocity field (Figure 11.3.2) demonstrates that the liquid slows down in response to the increased viscosity, as expected. In this model, there is a large “mushy” region, in which the motion of the fluid gradually decreases. This is in contrast to the first model, in which a momentum source was applied and a more abrupt change in the fluid motion was observed.
Figure 11.3.1: Laminar Viscosity Generated by a User-Defined Function

Figure 11.3.2: Contours of Velocity Magnitude Resulting from a User-Defined Viscosity
Figure 11.3.3: Stream Function Contours Suggesting Solidification
11.4 Reaction Rates

This section contains an application of a reaction rate UDF. It is executed as a compiled UDF in FLUENT.

11.4.1 A Custom Volume Reaction Rate

As an example of a compiled UDF, a custom volume reaction rate for a simple system of two gaseous species is considered. The species are named \textit{species-a} and \textit{species-b}. The reaction rate is one that converts \textit{species-a} into \textit{species-b} at a rate given by the following expression:

\[ R = \frac{K_1 X_a}{(1 + K_2 X_a)^2} \]  \hspace{1cm} (11.4-1)

where \(X_a\) is the mass fraction of \textit{species-a}, and \(K_1\) and \(K_2\) are constants.

The 2D (planar) domain consists of a 90-degree bend. The duct is 16 inches wide and approximately 114 inches long. A 6-inch-thick porous region covers the bottom and right-hand wall, and the reaction takes place in the porous region only. The species in the duct have identical properties. The density is 1.0 kg/m\(^3\), and the viscosity is \(1.72 \times 10^{-5}\) kg/m-s.

The outline of the domain is shown in Figure 11.4.1. The porous medium is the region below and to the right of the line that extends from the inlet on the left to the pressure outlet at the top of the domain.

Through the inlet on the left, gas that is purely \textit{species-a} enters with an \(x\) velocity of 0.1 m/s. The gas enters both the open region on the top of the porous medium and the porous medium itself, where there is an inertial resistance of 5 m\(^{-1}\) in each of the two coordinate directions. The laminar flow field (Figure 11.4.2) shows that most of the gas is diverted from the porous region into the open region.

The flow pattern is further substantiated by the vector plot shown in Figure 11.4.3. The flow in the porous region is considerably slower than that in the open region.

The source code (\texttt{rate.c}) that contains the UDF used to model the reaction taking place in the porous region is shown below. The function, named \texttt{vol_reac_rate}, is defined on a cell for a given species mass fraction using \texttt{DEFINE_VR_RATE}. The UDF performs a test to check for the porous region, and only applies the reaction rate equation to the porous region. The macro \texttt{FLUID THREAD P(t)} is used to determine if a cell thread is a fluid (rather than a solid) thread. The variable \texttt{THREAD VAR(t).fluid.porous} is used to check if a fluid cell thread is a porous region.
Figure 11.4.1: The Outline of the 2D Duct

Figure 11.4.2: Streamlines for the 2D Duct with a Porous Region
Figure 11.4.3: Velocity Vectors for the 2D Duct with a Porous Region
This UDF is executed as a compiled UDF in FLUENT. Follow the procedure for compiling source files using the Compiled UDFs panel that is described in Section 7.3.

Once the function vol_reac_rate is compiled and loaded, you can hook the reaction rate UDF to FLUENT by selecting the function’s name in the Volume Reaction Rate Function drop-down list in the User-Defined Function Hooks panel (Figure 8.2.24).

Define → User-Defined → Function Hooks...
11.4 Reaction Rates

Initialize and run the calculation. The converged solution for the mass fraction of species-a is shown in Figure 11.4.4. The gas that moves through the porous region is gradually converted to species-b in the horizontal section of the duct. No reaction takes place in the fluid region, although some diffusion of species-b out of the porous region is suggested by the wide transition layer between the regions of 100% and 0% species-a.

Figure 11.4.4: Mass Fraction for species-a Governed by a Reaction in a Porous Region
11.5 User-Defined Scalars

This section contains examples of UDFs that are used in conjunction with a user-defined scalar transport equation. See Chapter 10 for information on modeling user-defined scalar transport equations in FLUENT.

11.5.1 Postprocessing Using User-Defined Scalars

Below is an example of a compiled UDF that computes the gradient of temperature to the fourth power, and stores its magnitude in a user-defined scalar. The computed temperature gradient can, for example, be subsequently used to plot contours. Although the practical application of this UDF is questionable, its purpose here is to show the methodology of computing gradients of arbitrary quantities that can be used for postprocessing.

```c
#include "udf.h"

// UDF for computing the magnitude of the gradient of T^4

#define ADJUST(adjust_fcn, domain)
{
    Thread *t;
    cell_t c;
    face_t f;

    /* Make sure there are enough user-defined scalars. */
    if (n_uds < N_REQUIRED_UDS)
        Internal_Error("not enough user-defined scalars allocated");
```

© Fluent Inc. January 22, 2003
/* Fill first UDS with temperature raised to fourth power. */
thread_loop_c (t, domain)
{
  if (NULL != THREAD_STORAGE(t, SV_UDS_I(T4)))
  {
    begin_c_loop (c, t)
    {
      real T = C_T(c, t);
      C_UDSI(c, t, T4) = pow(T, 4.);
    }
    end_c_loop (c, t)
  }
}

/* Fill second UDS with magnitude of gradient. */
thread_loop_c (t, domain)
{
  if (NULL != THREAD_STORAGE(t, SV_UDS_I(T4)) &&
      NULL != T_STORAGE_R_NV(t, SV_UDSI_G(T4)))
  {
    begin_c_loop (c, t)
    {
      C_UDSI(c, t, MAG_GRAD_T4) = NV_MAG(C_UDSI_G(c, t, T4));
    }
    end_c_loop (c, t)
  }
}
thread_loop_f (t,domain)
{
    if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)) &&
        NULL != T_STORAGE_R_NV(t->t0,SV_UDSI_G(T4)))
    {
        begin_f_loop (f,t)
        {
            F_UDSI(f,t,MAG_GRAD_T4)=C_UDSI(F_C0(f,t),t->t0,MAG_GRAD_T4);
        }
        end_f_loop (f,t)
    }
}

The conditional statement if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4))) is used to check if the storage for the user-defined scalar with index T4 has been allocated, while NULL != T_STORAGE_R_NV(t->t0,SV_UDSI_G(T4)) checks whether the storage of the gradient of the user-defined scalar with index T4 has been allocated.

In addition to compiling this UDF, as described in Chapter 7, you will need to enable the solution of a user-defined scalar transport equation in FLUENT.

Define—User-Defined— Scalars...

See Section 10.3 for more details.
11.5.2 Implementing FLUENT’s P-1 Radiation Model

This section provides an example that demonstrates how the P1 radiation model can be implemented using UDFs and a user-defined scalar transport equation. In the P1 model, the variation of the incident radiation, \( G \), in the domain can be described by an equation that consists of a diffusion and source term.

The transport equation for incident radiation, \( G \), is given by Equation 11.5-1. The diffusion coefficient, \( \Gamma \), is given by Equation 11.5-2 and the source term is given by Equation 11.5-3. These equations are discussed in more detail in section 11.3.4 of the User’s Guide.

\[
\nabla \cdot (\Gamma \nabla G) + S^G = 0 \tag{11.5-1}
\]

\[
\Gamma = \frac{1}{3a + (3 - C) \sigma_s} \tag{11.5-2}
\]

\[
S^G = a \left( 4\sigma T^4 - G \right) \tag{11.5-3}
\]

As shown in the User’s Guide, the boundary condition for \( G \) at the walls is equal to the negative of the radiative wall heat flux, \( q_{r,w} \) (Equation 11.5-4), where \( \vec{n} \) is the outward normal vector. The radiative wall heat flux can be given by Equation 11.5-5.

\[
q_r \cdot \vec{n} = -\Gamma \nabla G \cdot \vec{n} \tag{11.5-4}
\]

\[
q_{r,w} = -\frac{\epsilon_w}{2(2 - \epsilon_w)} \left( 4\sigma T^4_w - G_w \right) \tag{11.5-5}
\]

This form of the boundary condition is unfortunately specified in terms of the incident radiation at the wall, \( G_w \). This mixed boundary condition can be avoided by solving first for \( G_w \) using Equations 11.5-4 and 11.5-5, resulting in Equation 11.5-6. Then, this expression for \( G_w \) is substituted back into Equation 11.5-5 to give the radiative wall heat flux \( q_{r,w} \) as Equation 11.5-7.

\[
G_w = \frac{4\sigma T^4_w E_w + \alpha_0 \Gamma_0}{A} [G_0 - \beta_0(G)] \tag{11.5-6}
\]

\[
q_r = -\frac{\alpha_0 \Gamma_0 E_w}{A \left( E_w + \alpha_0 \Gamma_0 \right)} \left[ 4\pi I_b(T_{iw}) - G_0 + \beta_0(G) \right] \tag{11.5-7}
\]

The additional \( \beta_0 \) and \( G_0 \) terms that appear in Equations 11.5-6 and 11.5-7 are a result of the evaluation of the gradient of incident radiation in Equation 11.5-4.
In FLUENT, the component of a gradient of a scalar directed normal to a cell boundary (face), $\nabla G \cdot n$, is estimated as the sum of primary and secondary components. The primary component represents the gradient in the direction defined by the cell centroids, and the secondary component is in the direction along the face separating the two cells. From this information, the face normal component can be determined. The secondary component of the gradient can be found using the Fluent macro `BOUNDARY_SECONDARY_GRADIENT_SOURCE`. The use of this macro first requires that cell geometry information be defined, which can be readily obtained by the use of a second macro, `BOUNDARY_FACE_GEOMETRY` (see Section 5.2.2). You will see these macros called in the UDF that defines the wall boundary condition for $G$.

To complete the implementation of the P1 model, the radiation energy equation must be coupled with the thermal energy equation. This is accomplished by modifying the source term and wall boundary condition of the energy equation. Consider first how the energy equation source term must be modified. The gradient of the incident radiation is proportional to the radiative heat flux. A local increase (or decrease) in the radiative heat flux is attributable to a local decrease (or increase) in thermal energy via the absorption and emission mechanisms. The gradient of the radiative heat flux is therefore a (negative) source of thermal energy. As shown in the User’s Guide, the source term for the incident radiation Equation 11.5-3 is equal to the gradient of the radiative heat flux and hence its negative specifies the source term needed to modify the energy equation.

Now consider how the energy boundary condition at the wall must be modified. Locally, the only mode of energy transfer from the wall to the fluid that is accounted for by default is conduction. With the inclusion of radiation effects, radiative heat transfer to and from the wall must also be accounted for. (This is done automatically if you use FLUENT’s built-in P1 model.) The `DEFINE_HEAT_FLUX` macro allows the wall boundary condition to be modified to accommodate this second mode of heat transfer by specifying the coefficients of the $qir$ equation discussed in Section 4.3.7. The net radiative heat flux to the wall has already been given as Equation 11.5-5. Comparing this equation with that for $qir$ in Section 4.3.7 will result in the proper coefficients for $cir$.

In this example, the implementation of the P1 model can be accomplished through six separate UDFs. They are all included in a single source file, which can be executed as a compiled UDF. The single user-defined scalar transport equation for incident radiation, $G$, uses a `DEFINE_DIFFUSIVITY` UDF to define $\Gamma$ of Equation 11.5-2, and a `DEFINE_SOURCE` UDF to define the source term of Equation 11.5-3. The boundary condition for $G$ at the walls is handled by assigning, in `DEFINE_PROFILE`, the negative of Equation 11.5-7 as the specified flux. A `DEFINE_ADJUST` UDF is used to instruct FLUENT to check that the proper number of user-defined scalars has been defined (in the solver). Lastly, the energy equation must be assigned a source term equal to the negative of that used in the incident radiation equation and the `DEFINE_HEAT_FLUX` UDF is used to alter the boundary conditions at the walls for the energy equation.
In the solver, at least one user-defined scalar equation must be enabled. The scalar diffusivity is assigned in the Materials panel for the scalar equation. The scalar source and energy source terms are assigned in the boundary condition panel for the fluid zones. The boundary condition for the scalar equation at the walls is assigned in the boundary condition panel for the wall zones. The DEFINE_ADJUST and DEFINE_HEAT_FLUX functions are assigned in the User-Defined Function Hooks panel.

```c
/* Implementation of the P1 model using user-defined scalars */
#include "udf.h"
enum
{
   P1,
   N_REQUIRED_UDS
};

static real abs_coeff = 1.0; /* absorption coefficient */
static real scat_coeff = 0.0; /* scattering coefficient */
static real las_coeff = 0.0; /* linear-anisotropic scattering coefficient */
static real epsilon_w = 1.0; /* wall emissivity */

DEFINE_ADJUST(p1_adjust, domain)
{
   /* Make sure there are enough user defined-scalars. */
   if (n_uds < N_REQUIRED_UDS)
      Internal_Error("not enough user-defined scalars allocated");
}

DEFINE_SOURCE(energy_source, c, t, dS, eqn)
{
   dS[eqn] = -16.*abs_coeff*SIGMA_SBC*pow(C_T(c,t),3.);
   return -abs_coeff*(4.*SIGMA_SBC*sig_C_T(c,t) - C_UDSI(c,t,P1));
}
```
DEFINE_SOURCE(p1_source, c, t, dS, eqn)
{
    dS[eqn] = -abs_coeff;
    return abs_coeff*(4.*SIGMA_SBC*pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
}

DEFINE_DIFFUSIVITY(p1_diffusivity, c, t, i)
{
    return 1./(3.*abs_coeff + (3. - las_coeff)*scat_coeff);
}

DEFINE_PROFILE(p1_bc, thread, position)
{
    face_t f;
    real A[ND_ND],At;
    real dG[ND_ND],dr0[ND_ND],es[ND_ND],ds,A_by_es;
    real aterm,alpha0,beta0,gamma0,Gsource,Ibw;
    real Ew = epsilon_w/(2.*(2. - epsilon_w));
    Thread *t0=thread->t0;
    
    /* Do nothing if areas aren’t computed yet or not next to fluid. */
    if (!Data_Valid_P() || !FLUID_THREAD_P(t0)) return;
    
    begin_f_loop (f,thread)
    {
        cell_t c0 = F_C0(f,thread);

        BOUNDARY_FACE_GEOMETRY(f,thread,A,ds,es,A_by_es,dr0);
        At = NV_MAG(A);

        if (NULLP(T_STORAGE_R_NV(t0,SV_UDSI_G(P1))))
            Gsource = 0.; /* if gradient not stored yet */
        else
            BOUNDARY_SECONDARY_GRADIENT_SOURCE(Gsource,SV_UDSI_G(P1),
                dG,es,A_by_es,1.);

            gamma0 = C_UDSI_DIFF(c0,t0,P1);
            alpha0 = A_by_es/ds;
            beta0  = Gsource/alpha0;
            aterm  = alpha0*gamma0/At;
    }
}
Ibw = SIGMA_SBC*pow(WALL_TEMP_OUTER(f,thread),4.)/M_PI;

/* Specify the radiative heat flux. */
F_PROFILE(f,thread,position) =
aterm*Ew/(Ew + aterm)*(4.*M_PI*Ibw - C_UDSI(c0,t0,P1) + beta0);
}
end_f_loop (f,thread)
}

DEFINE_HEAT_FLUX(heat_flux, f, t, c0, t0, cid, cir)
{
    real Ew = epsilon_w/(2.*(2. - epsilon_w));

    cid[0] = Ew * F_UDSI(f,t,P1);
    cid[3] = 4.0 * Ew * SIGMA_SBC;
}

Appendix A. DEFINE Macro Definitions

A.1 General Solver DEFINE Macros

The following definitions for general solver DEFINE macros (see Section 4.2) are taken from the udf.h header file.

#define DEFINE_ADJUST(name, domain) void name(Domain *domain)

#define DEFINE_EXECUTE_AT_END(name) void name(void)

#define DEFINE_INIT(name, domain) void name(Domain *domain)

#define DEFINE_ON_DEMAND(name) void name(void)

#define DEFINE_RW_FILE(name, fp) void name(FILE *fp)
A.2 Model-Specific DEFINE Macros

The following definitions for model-specific DEFINE macros (see Section 4.3) are taken from the udf.h header file.

```c
#define DEFINE_CHEM_STEP(name, ifail, n, dt, p, temp, yk) 
   void name(int *ifail, int n, real dt, real *p, real *temp, real *yk)

#define DEFINE_DIFFUSIVITY(name, c, t, i) 
   real name(cell_t c, Thread *t, int i)

#define DEFINE_DOM_DIFFUSE_REFLECTIVITY(name, t, nb, n_a, n_b, diff_ref_a, \ 
   diff_tran_a, diff_ref_b, diff_tran_b) 
   void name(Thread *t, int nb, real n_a, real n_b, real *diff_ref_a, \ 
   real *diff_tran_a, real *diff_ref_b, real *diff_tran_b)

#define DEFINE_DOM_SPECULAR_REFLECTIVITY(name, f, t, nb, n_a, n_b, \ 
   ray_direction, e_n, total_internal_reflection, \ 
   specular_reflectivity, specular_transmissivity) 
   void name(face_t f, Thread *t, int nb, real n_a, real n_b, \ 
   real ray_direction[], real e_n[], \ 
   int *total_internal_reflection, real *specular_reflectivity,\ 
   real *specular_transmissivity)

#define DEFINE_DOM_SOURCE(name, c, t, ni, nb, emission, in_scattering, \ 
   abs_coeff, scat_coeff) 
   void name(cell_t c, Thread* t, int ni, int nb, real *emission, \ 
   real *in_scattering, real *abs_coeff, real *scat_coeff)

#define DEFINE_HEAT_FLUX(name, f, t, c0, t0, cid, cir) 
   void name(face_t f, Thread *t, cell_t c0, Thread *t0, \ 
   real cid[], real cir[])

#define DEFINE_NET_REACTION_RATE(name, p, temp, yi, rr, jac) 
   void name(double *p, double *temp, double *yi, double *rr, \ 
   double *jac)

#define DEFINE_NOX_RATE(name, c, t, NOx) 
   void name(cell_t c, Thread *t, NOx_Parameter *NOx)

#define DEFINE_PRANDTL_K(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_D(name, c, t) real name(cell_t c, Thread *t)
```
#define DEFINE_PRANDTL_O(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_T(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_T_WALL(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PROFILE(name, t, i) void name(Thread *t, int i)

#define DEFINE_PROPERTY(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PR_RATE(name, c, t, r, mw, ci, p, sf, dif_index, 
        cat_index, rr) 
    void name(cell_t c, Thread *t, Reaction *r, real *mw, real *ci, 
             Tracked_Particle *p, real *sf, int dif_index, 
             int cat_index, real *rr)

#define DEFINE_SCAT_PHASE_FUNC(name, c, f) 
    real name(real c, real *f)

#define DEFINE_SOURCE(name, c, t, dS, i) 
    real name(cell_t c, Thread *t, real dS[], int i)

#define DEFINE_SR_RATE(name, f, t, r, mw, yi, rr) 
    void name(face_t f, Thread *t, 
              Reaction *r, real *mw, real *yi, real *rr)

#define DEFINE_TURB_PREMIX_SOURCE(name, c, t, 
                                   turbulent_flame_speed, source) 
    void name(cell_t c, Thread *t, real *turbulent_flame_speed, 
              real *source)

#define DEFINE_TURBULENT_VISCOSITY(name, c, t) 
    real name(cell_t c, Thread *t)

#define DEFINE_UDS_FLUX(name, f, t, i) real name(face_t f, 
                                           Thread *t, int i)

#define DEFINE_UDS_UNSTEADY(name, c, t, i, apu, su) 
    void name(cell_t c, Thread *t, int i, real *apu, real *su)

#define DEFINE_VR_RATE(name, c, t, r, mw, yi, rr, rr_t) 
    void name(cell_t c, Thread *t, 
              Reaction *r, real *mw, real *yi, real *rr, real *rr_t)
A.3 Multiphase DEFINE Macros

The following definitions for multiphase DEFINE macros (see Section 4.4) are taken from the udf.h header file.

#define DEFINE_CAVITATION_RATE(name, c, t, p, rhoV, rhoL, vofV, p_v, n_b, m_dot) \
    void name(cell_t c, Thread *t, real *p, real *rhoV, real *rhoL, \ 
    real *vofV, real *p_v, real *n_b, real *m_dot)

#define DEFINE_EXCHANGE_PROPERTY(name, c, mixture_thread, \ 
    second_column_phase_index, first_column_phase_index) \ 
    real name(cell_t c, Thread *mixture_thread, \ 
    int second_column_phase_index, int first_column_phase_index)

#define DEFINE_VECTOR_EXCHANGE_PROPERTY(name, c, mixture_thread, \ 
    second_column_phase_index, first_column_phase_index, vector_result) \ 
    void name(cell_t c, Thread *mixture_thread, \ 
    int second_column_phase_index, \ 
    int first_column_phase_index, real *vector_result)
A.4 Dynamic Mesh Model DEFINE Macros

The following definitions for dynamic mesh model DEFINE macros (see Section 4.5) are taken from the udf.h header file.

#define DEFINE_CG_MOTION(name, dt, vel, omega, time, dtime) \
    void name(Dynamic_Thread *dt, real vel[], real omega[], real time,\n            real dtime)

#define DEFINE_GEOM(name, d, dt, position) \
    void name(Domain *d, Dynamic_Thread *dt, real *position)

#define DEFINE_GRID_MOTION(name, d, dt, time, dtime) \
    void name(Domain *d, Dynamic_Thread *dt, real time, real dtime)
A.5 Discrete Phase Model DEFINE Macros

The following definitions for DPM DEFINE macros (see Section 4.6) are taken from the dpm.h header file. Note that dpm.h is included in the udf.h header file.

#define DEFINE_DPM_BC(name, p, t, f, normal, dim) 
   int name(Tracked_Particle *p, Thread *t, face_t f, 
               real normal[], int dim)

#define DEFINE_DPM_BODY_FORCE(name, p, i) 
   real name(Tracked_Particle *p, int i)

#define DEFINE_DPM_DRAG(name, Re, p) 
   real name(real Re, Tracked_Particle *p)

#define DEFINE_DPM_EROSION(name, p, t, f, normal, alpha, Vmag, mdot) 
   void name(Tracked_Particle *p, Thread *t, face_t f, real normal[], 
              real alpha, real Vmag, real mdot)

#define DEFINE_DPM_INJECTION_INIT(name, I) void name(Injection *I)

#define DEFINE_DPM_LAW(name, p, ci) 
   void name(Tracked_Particle *p, int ci)

#define DEFINE_DPM_OUTPUT(name, header, fp, p, t, plane) 
   void name(int header, FILE *fp, Tracked_Particle *p, 
              Thread *t, Plane *plane)

#define DEFINE_DPM_PROPERTY(name, c, t, p) 
   real name(cell_t c, Thread *t, Tracked_Particle *p)

#define DEFINE_DPM_SCALAR_UPDATE(name, c, t, initialize, p) 
   void name(cell_t c, Thread *t, int initialize, Tracked_Particle *p)

#define DEFINE_DPM_SOURCE(name, c, t, S, strength, p) 
   void name(cell_t c, Thread *t, dpms_t *S, real strength, 
             Tracked_Particle *p)

#define DEFINE_DPM_SPRAY_COLLIDE(name, tp, p) 
   void name(Tracked_Particle *tp, Particle *p)

#define DEFINE_DPM_SWITCH(name, p, ci) 
   void name(Tracked_Particle *p, int ci)
Bibliography


Index

A[ND_ND], 5-7
absorption and scattering              
    coefficient UDFs, 4-63
accessing domain pointer not          
    passed as argument, 6-11
accretion rate UDF, 4-130
adjacent cell index macros, 5-8
adjacent cell thread macros, 5-9
adjust UDF, 4-3
advection term, 4-87
advective flux, 4-87
advective flux field, 4-88
ANSI C preprocessor, 7-5
area normal vector macro, 5-7
arguments, 5-2
arithmetic operators, 2-12
arrays, 2-8
ARRH, 4-39, 5-28
Arrhenius constants, 4-77

begin...end_c_loop, 6-3, 9-22
begin...end_c_loop_ext, 9-22
begin...end_c_loop_int, 9-22
begin...end_f_loop, 6-4, 9-22
begin_c_loop_all, 4-8
body force UDF, 4-126
boiling point, 4-143
boundary condition UDF
    examples, 11-1, 11-2
    for DPM, 4-118
    general-purpose, 4-53
Boundary Conditions panel, 10-8
boundary face area normals, direction, 5-5
boundary face, partition, 9-9
boundary zone faces, partitioned grid, 9-29

BOUNDARY_FACE_GEOMETRY, 5-7, 11-32
BOUNDARY_FACE_THREAD_P, 4-89, 5-8
BOUNDARY_SECONDARY_GRADIENT_SOURCE, 11-32
building a shared library, 7-2, 7-9
C compiler, 7-2
C preprocessor, 7-1, 7-5
C programming
    arrays, 2-8
    casting, 2-7
    commenting code, 2-2
    constants, 2-2
    control statements, 2-10
        for loops, 2-11
        if, 2-10
        if-else, 2-10
    data types, 2-2
    #define, 2-16
    example, 3-4
    file inclusion, 2-17
FORTRAN, comparison with, 2-17
functions, 2-7, 2-13
    I/O, 2-14
    input-output, 2-14
    mathematical, 2-13
    trigonometric, 2-13
#include, 2-17
macro substitution, 2-16
operators, 2-12
    arithmetic, 2-12
    logical, 2-12
pointers, 2-8
    as function arguments, 2-9
Index

variables, 2-3
  declaring, 2-4
  external, 2-4
  global, 2-3
  local, 2-3
  static, 2-6
typedef, 2-7
C_CENTROID, 4-8, 4-9, 4-74, 5-4, 6-6
C_CP, 4-103
C_D, 4-47, 4-85, 5-9
C_DP, 4-106
C_FACE, 4-136, 5-9, 6-5
c_face_loop, 4-77, 4-78, 6-4
C_FACE_THREAD, 4-77, 4-78, 4-136,
  5-9, 6-5
C_FMEAN, 4-82
C_H, 5-9
C_K, 4-47, 4-85, 5-9
C_KL, 4-103
C_MU_EFF, 4-24
C_MU_L, 4-47, 4-101, 4-103
C_MU_T, 4-47
C_NFACES, 5-4
C_NNODES, 5-4
C_NODE, 6-5
c_node_loop, 6-5
C_NUT, 5-9
C_O, 5-9
C_P, 5-9
C_PHASE_DIAMETER, 4-103
C_R, 4-24, 4-78, 4-85, 4-88, 4-94,
  4-101, 4-103, 4-106, 5-9
C_R_M1, 4-92
C_STORAGE_R, 4-92
C_T, 4-10, 4-67, 4-78, 4-104, 4-156,
  5-9, 11-18
C_U, 4-101, 4-103, 5-9
C_UDMI, 4-10, 4-131, 5-18, 8-7
C_UDSI, 4-5, 5-15
C_V, 4-101, 4-103, 5-9
C_VOF, 4-101, 4-103, 4-104, 6-6
C_VOLUME, 4-4, 4-7, 4-10, 4-92, 5-6
C_W, 4-103, 5-9

C_YI, 4-67, 4-136, 5-9
case file functions, reading
  and writing, 4-13
casting, 2-7
cavitation rate UDF, 4-98
cell, 1-8, 1-9
  neighboring thread
    variables, accessing, 1-8
    values, checking accessibility, 6-27
cell centroid, 5-6
cell centroid macro, 5-2, 5-4
cell face macro, 5-9
cell face thread macro, 5-9
cell gradient macros, 5-11
cell identifier, 5-2
cell looping macro, general-purpose, 9-25
cell looping macros, 6-1
cell looping macros, parallel, 9-22
cell partition IDs, parallel, 9-28
cell reconstruction gradient macros, 5-11
cell temperature macro, 5-2
cell variables
  macros, 5-9
cell volume macro, 5-6
cell_t, 1-10
center of gravity motion UDF, 4-108
chemistry step UDF, 4-19
coal emissivity, 4-144
coal scattering, 4-144
combustion applications, 10-1
communication macros, parallel, 9-15, 9-16
Compiled UDFs panel, 7-2, 7-7
compiled UDFs, 1-2, 1-6, 7-2, 7-7
  building shared library, 7-2
  example, 11-23, 11-31
  loading shared library, 7-2
  shared library, 1-6
  Windows parallel network, 7-7, 7-9
writing case files, 7-10
compiler directives
  about, 9-12
  example, 9-13
compiling source files about, 7-7
    procedure, 7-7
compute nodes, 9-4
connectivity macros, 5-4
control statements, 2-10
convective flux, 10-1
coupled solver, 1-13
cpp, 7-5
CPP Command Name, 7-5
CURRENT_TIME, 4-21, 6-22, 11-7
CURRENT_TIMESTEP, 6-22
custom law UDFs for DPM, 4-139
Custom Laws panel, 8-51, 8-58
data file functions, reading and writing, 4-13
data types, 1-10
    user-defined, 2-7
data, accessing, 1-16
Data_Valid_P, 4-5, 4-109, 6-27
dsegment your UDF, 7-6
#define, 2-16
DEFINE macros, 1-4, 4-1, 6-14
    DPM, 4-116
        dynamic mesh, 4-107
        format, 1-4
        general solver, 4-2
        model-specific, 4-15
        multiphase, 1-16, 4-97
            Eulerian model, 1-19
            mixture model, 1-18
            VOF model, 1-17
DEFINE_ADJUST UDF
    defining, 4-3
    example, 11-32
    hooking to FLUENT, 8-2
DEFINE_CAVITATION_RATE UDF
    defining, 4-98
    hooking to FLUENT, 8-32
DEFINE.CG.MOTION UDF
    defining, 4-108
    hooking to FLUENT, 8-38
DEFINE_CHEM_STEP UDF
    defining, 4-19
    hooking to FLUENT, 8-8
DEFINE_DELTAT UDF
    defining, 4-21
    hooking to FLUENT, 8-9
DEFINE_DIFFUSIVITY UDF
    defining, 4-23
    example, 11-32
    hooking to FLUENT, 8-10
DEFINE_DOM_DIFFUSE_REFLECTIVITY UDF
    defining, 4-25
    hooking to FLUENT, 8-12
DEFINE_DOM_SOURCE UDF
    defining, 4-27
    hooking to FLUENT, 8-13
DEFINE_DOM_SPECULAR_REFLECTIVITY UDF
    defining, 4-29
    hooking to FLUENT, 8-14
DEFINE_DPM_Bc UDF
    defining, 4-118
    hooking to FLUENT, 8-44
DEFINE_DPM_BODY_FORCE UDF
    defining, 4-126
    hooking to FLUENT, 8-46
DEFINE_DPM_DRAG UDF
    defining, 4-128
    hooking to FLUENT, 8-47
DEFINE_DPM_EROSION UDF
    defining, 4-130
    hooking to FLUENT, 8-48
DEFINE_DPM_INJECTION_INIT UDF
    defining, 4-136
    hooking to FLUENT, 8-50
DEFINE_DPM_LAW UDF
    defining, 4-139
    hooking to FLUENT, 8-51
DEFINE_DPM_OUTPUT UDF
    defining, 4-141
    hooking to FLUENT, 8-52
DEFINE_DPMPROPERTY UDF
defining, 4-143
hooking to FLUENT, 8-53
DEFINE_DPSCALAR_UPDATE UDF
defining, 4-147
hooking to FLUENT, 8-55
DEFINE_DPSOURCE UDF
defining, 4-151
hooking to FLUENT, 8-56
DEFINE_DPMSPRAY_COLLIDE UDF
defining, 4-153
hooking to FLUENT, 8-57
DEFINE_DPSWITCH UDF
defining, 4-156
hooking to FLUENT, 8-58
DEFINE_EXCHANGEPROPERTY UDF
defining, 4-100
hooking to FLUENT, 8-34
DEFINE_EXECUTE_AT_END UDF
defining, 4-6
hooking to FLUENT, 8-3
DEFINE_GEOM UDF
defining, 4-111
hooking to FLUENT, 8-40
DEFINE_GRID_MOTION UDF
defining, 4-113
hooking to FLUENT, 8-42
DEFINE_HEAT_FLUX UDF
defining, 4-32
example, 11-32
hooking to FLUENT, 8-15
DEFINE_INIT UDF
defining, 4-8
hooking to FLUENT, 8-4
DEFINE_NETREACTION_RATE UDF
defining, 4-34
hooking to FLUENT, 8-16
DEFINE_NOX_RATE UDF
defining, 4-36
hooking to FLUENT, 8-17
DEFINE_ON_DEMAND UDF
defining, 4-10
hooking to FLUENT, 8-5
DEFINE_PR_RATE UDF
defining, 4-40
hooking to FLUENT, 8-18
DEFINE_PRANDTL UDFs
defining, 4-46
hooking to FLUENT, 8-19
DEFINE_PROFILE UDF
defining, 4-53
example, 11-4, 11-32
hooking to FLUENT, 8-20
DEFINE_PROPERTY UDF
defining, 4-63
example, 11-18
hooking to FLUENT, 8-21
DEFINE_RW_FILE UDF
defining, 4-13
hooking to FLUENT, 8-6
DEFINE_SCAT_PHASE_FUNC UDF
defining, 4-69
hooking to FLUENT, 8-23
DEFINE_SOURCE UDF
defining, 4-72
example, 11-13, 11-32
hooking to FLUENT, 8-25
DEFINE_SR_RATE UDF
defining, 4-76
hooking to FLUENT, 8-26
DEFINE_TURB_PREMIX_SOURCE UDF
defining, 4-82
hooking to FLUENT, 8-27
DEFINE_TURBULENT_VISCOSITY UDF
defining, 4-85
hooking to FLUENT, 8-28
DEFINE_UDS_FLUX UDF
defining, 4-87
example, 10-7
hooking to FLUENT, 8-29
DEFINE_UDS_UNSTEADY UDF
defining, 4-91
example, 10-8
hooking to FLUENT, 8-30
Index

DEFINE_VECTOR_EXCHANGE_PROPERTY UDF
- defining, 4-105
- hooking to FLUENT, 8-36

DEFINE_VR_RATE UDF
- defining, 4-93
- example, 11-23
- hooking to FLUENT, 8-31

DEFINE macros, 4-1
- deforming zone geometry UDF, 4-111
- demo_calc, 4-14
- density UDF, 4-63, 8-22
- derivative variable macros, 5-15
- derivatives, source term, 4-72
- diffuse reflectivity UDF, 4-25
- diffusion coefficient, 5-6, 10-1
- diffusive flux, 5-6
- discrete ordinates model UDFs
  - diffuse reflectivity, 4-25
  - scattering phase, 4-69
  - source terms, 4-27
  - specular reflectivity, 4-29

Discrete Phase Model panel, 8-46-8-48, 8-55, 8-56
- discrete phase model UDFs
  - body force, 4-126
  - boundary conditions, 4-118
  - custom particle laws, 4-139
  - defining, 4-116
  - drag coefficient, 4-128
  - erosion and accretion rates, 4-130
  - for material properties, 4-143
  - for sampling device output, 4-141
  - for switching custom laws, 4-156
  - hooking to FLUENT, 8-44
  - particle initialization, 4-136
  - scalar update, 4-147
  - source term, 4-151
  - spray collide, 4-153
- dispersed phase properties, 4-143

Display Assembly Listing, 7-6

Domain, 1-10
- domain
  - pointer, 1-11
- domain ID, 6-17
- DOMAIN_ID, 6-6, 6-17
- domain_id, 1-12, 6-11
- DOMAIN_SUB_DOMAIN, 6-13, 6-14
- DOMAIN_SUPER_DOMAIN, 6-15
- domains, 1-9
  - interaction, 1-11
  - mixture, 1-11
  - multiphase UDFs, 1-16
  - phase, 1-11
  - referencing, 1-16
  - subdomains, 1-11
  - superdomains, 1-11

DPM DEFINE macros
- quick reference guide, 4-116

DPM macros
- particle species, laws,
  - and user scalars, 5-26
- particles at current position, 5-24
- particles at injection into domain, 5-25

DPM property UDF, 4-143

DPM variable macros, 5-24
dpm.h file, 5-24

DPM_OUTPUT, 4-148
- drag coefficient UDF, multiphase, 4-100
- drag law, default, 4-101

DT_THREAD, 4-109, 4-114
- duct flow, 11-12
- dynamic loading, 7-2
dynamic mesh DEFINE macros
- quick reference guide, 4-107
dynamic mesh UDFs
  - center of gravity motion, 4-108
  - deforming zone geometry, 4-111
  - grid motion, 4-113

Dynamic Zones panel, 8-38, 8-40, 8-42
edge, 1-9
emission term, 4-28
erosion rate UDF, 4-130
**Error**, 6-27
error messages, 7-6
errors
  - compiling source files, 7-11
  - interpreting source files, 7-6
  - parallel, 7-7, 7-11
Eulerian model
  - dispersed turbulence model, 1-21
  - laminar flow, 1-19
  - mixture turbulence model, 1-20
  - per-phase turbulence model, 2-1
examples, UDF
  - compiled only, 4-43, 4-48, 4-77, 4-78, 4-82, 4-89, 4-94, 4-108, 4-111, 4-113, 4-114, 4-118, 4-121, 4-126, 4-128, 4-131, 4-136, 4-144, 4-148, 4-156, 11-28
  - interpreted or compiled, 4-4, 4-7, 4-8, 4-10, 4-13, 4-24, 4-55, 4-56, 4-62, 4-68, 4-70, 4-74, 4-85, 4-92, 4-99, 4-140
exchange macros, parallel, 9-34
exchange property UDF, 4-100
**EXCHANGE_SVAR_FACE_MESSAGE**, 9-34
**EXCHANGE_SVAR_MESSAGE**, 9-34
**Execute On Demand** panel, 8-5
execute-at-end UDF, 4-6
exterior cell looping macro, parallel, 9-24
external emissivity profile, 4-53

**F_AREA**, 4-77, 4-78, 4-89, 4-109, 4-131, 4-136, 5-5, 9-9, 9-36
**F_C0**, 4-77, 4-78, 4-88, 4-131, 5-8
**F_C1**, 4-88, 5-8
**F_CENTROID**, 4-55, 4-56, 4-60, 4-118, 5-4, 6-10, 6-12, 6-17, 11-4
**F_D**, 5-21
**F_FLUX**, 4-88, 5-21
**F_H**, 5-21
**F_K**, 5-21
**F_NNODLES**, 5-4

**F_NODE**, 4-114
**f_node_loop**, 4-114
**F_P**, 5-21
**F_PART**, 9-29
**F_PROFILE**, 4-54–4-56, 4-60, 6-10, 11-4
**F_R**, 5-21
**F_STORAGE_R**, 4-130
**F_T**, 4-78, 5-21
**F_U**, 4-88, 5-21
**F_UDMI**, 4-131, 5-22, 8-7
**F_V**, 4-88, 5-21
**F_W**, 4-88, 5-21
**F_YI**, 5-21
face, 1-9
  - variables
    - setting, 6-10
facing area vector macro, 5-5
face centroid macro, 5-4
face identifier, 5-2
face looping macro, general-purpose, 9-26
face looping macros, 6-1
face looping utilities, 4-54
face normal gradient, 5-6
face partition IDs, parallel, 9-29
face variable macros, 5-21
**face_t**, 1-10
**fclose** function, 2-15
**fclose**, 9-39
**FILE**, 6-12
file inclusion, 2-17
**File XY Plot** panel, 11-11
**Fill_Face_Part_With_Different**, 9-29
**Fill_Face_Part_With_Same**, 9-29
flow variable macros, 5-9, 5-21
**FLUENT**
  - data types, 1-10
    - case-sensitive, 1-10
**FLUENT** variables
  - accessing, 5-1
**Fluid** panel, 8-25, 10-6, 10-10, 11-15
fluid thread, checking, 6-28
**FLUID_THREAD_P**, 4-5, 4-7, 4-131, 4-156, 6-28, 11-23
Index

flux function, 4-87
fopen function, 2-14
fopen, 9-39
for loops, 2-11
FORTRAN, 2-17
fprintf function, 2-15
fprintf, 4-13, 6-12
fscanf function, 2-16
fscanf, 4-13
functions, 2-7, 2-13
reader, 4-14
writer, 4-14
general solver DEFINE macros
quick reference guide, 4-2
general solver UDFs
DEFINE macros, 6-14
hooking to FLUENT, 8-2
generic_property, 4-67
global reduction macros, 9-17
gradient vector macros, 5-9
grid
components
domains, threads, cells,
faces, 1-8
topology, 1-8
zones, 1-8
grid motion UDF, 4-113
header files
udf.h, 7-2
heat flux function, 11-33
heat generation rate profile, 4-53
heat transfer coefficient UDF,
multiphase, 4-100
hooking UDFs to FLUENT
about, 8-1
DPM, 8-44
dynamic mesh, 8-38
general solver, 8-2
model-specific, 8-8
multiphase, 8-32
host_to_node, 9-15
I/O functions, 2-14
I_AM_NODE_HOST_P, 9-17
I_AM_NODE_LAST_P, 9-17
I_AM_NODE_LESS_P, 9-17
I_AM_NODE_MORE_P, 9-17
I_AM_NODE_ONE_P, 9-17
I_AM_NODESAME_P, 9-17
I_AM_NODEZERO_P, 9-17, 9-39
identifying processes, parallel, 9-35
if statement, 2-10
if-else statement, 2-10
incident radiation, 11-31
#include, 2-17
initialization UDF, 4-8
Injections panel, 8-50
input-output functions, 2-14
interaction domains, 1-11
interior cell looping macro, parallel, 9-23
interior faces, partitioned grid, 9-29
INTERIOR_FACE_GEOMETRY, 5-7
Interpreted UDFs panel, 3-5, 7-1,
7-3, 7-4, 7-6
interpreted UDFs, 1-2, 1-6, 1-7, 7-3
errors in interpreting source file, 7-6
example, 11-18
interpreter, 1-6
restrictions, 1-6
special considerations for parallel, 7-7
Windows parallel network, 7-3, 7-4
writing case files, 7-6
interpreted vs. compiled, 1-6
interpreting and compiling source files, 7-1
interpreting source files
about, 7-3
procedure, 7-3
Iterate panel, 8-9, 11-8
k-% turbulence model, 4-85
k-ω turbulence model, 4-85
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>laminar flow speed</td>
<td>4-63</td>
</tr>
<tr>
<td>LES turbulence model</td>
<td>4-85</td>
</tr>
<tr>
<td>libudf</td>
<td>7-2</td>
</tr>
<tr>
<td>lift coefficient UDF, multiphase</td>
<td>4-100</td>
</tr>
<tr>
<td>loading a shared library</td>
<td>7-2, 7-9</td>
</tr>
<tr>
<td>log file</td>
<td>7-6</td>
</tr>
<tr>
<td>logical operators</td>
<td>2-12</td>
</tr>
<tr>
<td>looking up a thread pointer</td>
<td>6-16</td>
</tr>
<tr>
<td><strong>Lookup_Thread</strong>, 6-16, 9-39</td>
<td></td>
</tr>
<tr>
<td>looping macros</td>
<td></td>
</tr>
<tr>
<td>for multiphase applications</td>
<td>6-6</td>
</tr>
<tr>
<td>for parallel</td>
<td>9-22</td>
</tr>
<tr>
<td>general-purpose</td>
<td>6-2</td>
</tr>
<tr>
<td>looping over cell threads in domain</td>
<td>6-2</td>
</tr>
<tr>
<td>looping over cells in thread</td>
<td>6-3</td>
</tr>
<tr>
<td>looping over exterior cells, parallel</td>
<td>9-24</td>
</tr>
<tr>
<td>looping over face threads in domain</td>
<td>6-3</td>
</tr>
<tr>
<td>looping over faces in thread</td>
<td>6-4</td>
</tr>
<tr>
<td>looping over faces on cell</td>
<td>6-4</td>
</tr>
<tr>
<td>looping over interior cells, parallel</td>
<td>9-23</td>
</tr>
<tr>
<td>looping over nodes of cell</td>
<td>6-5</td>
</tr>
<tr>
<td><strong>M_PI</strong>, 4-41, 4-44, 4-99, 4-121, 4-156, 6-29</td>
<td></td>
</tr>
<tr>
<td>macro substitution</td>
<td>2-16</td>
</tr>
<tr>
<td><strong>macros</strong></td>
<td></td>
</tr>
<tr>
<td>adjacent cell index</td>
<td>5-8</td>
</tr>
<tr>
<td>adjacent cell thread</td>
<td>5-9</td>
</tr>
<tr>
<td>area normal vector</td>
<td>5-7</td>
</tr>
<tr>
<td>cell diffusion coefficients</td>
<td>5-15</td>
</tr>
<tr>
<td>cell face</td>
<td>5-9</td>
</tr>
<tr>
<td>cell face thread</td>
<td>5-9</td>
</tr>
<tr>
<td>cell thermodynamic properties</td>
<td>5-15</td>
</tr>
<tr>
<td>cell variable</td>
<td>5-9</td>
</tr>
<tr>
<td>cell volume</td>
<td>5-6</td>
</tr>
<tr>
<td>centroid variable</td>
<td>5-6</td>
</tr>
<tr>
<td>centroid variables</td>
<td>5-4</td>
</tr>
<tr>
<td>derivative variable</td>
<td>5-15</td>
</tr>
<tr>
<td>DPM variable</td>
<td>5-24</td>
</tr>
<tr>
<td>error</td>
<td>6-26</td>
</tr>
<tr>
<td>face area vector</td>
<td>5-5</td>
</tr>
<tr>
<td>face variable</td>
<td>5-4, 5-21</td>
</tr>
<tr>
<td>flow variable</td>
<td>5-9, 5-21</td>
</tr>
<tr>
<td><strong>FLUENT</strong> variables, accessing</td>
<td>5-1</td>
</tr>
<tr>
<td>geometry</td>
<td>5-3</td>
</tr>
<tr>
<td>gradient vector</td>
<td>5-9</td>
</tr>
<tr>
<td>looping</td>
<td></td>
</tr>
<tr>
<td>general-purpose</td>
<td>6-2</td>
</tr>
<tr>
<td>multiphase-specific</td>
<td>6-6</td>
</tr>
<tr>
<td>material property</td>
<td>5-15</td>
</tr>
<tr>
<td>message</td>
<td>6-26</td>
</tr>
<tr>
<td>multiphase variables</td>
<td>5-23</td>
</tr>
<tr>
<td>node coordinates</td>
<td>5-3</td>
</tr>
<tr>
<td>node variable</td>
<td>5-4</td>
</tr>
<tr>
<td>NOx</td>
<td>5-28</td>
</tr>
<tr>
<td>particle variable</td>
<td>5-24</td>
</tr>
<tr>
<td>previous time step</td>
<td>5-11</td>
</tr>
<tr>
<td>reconstruction gradient vector</td>
<td>5-9</td>
</tr>
<tr>
<td>Reynolds Stress Model</td>
<td>5-15</td>
</tr>
<tr>
<td>user-defined memory</td>
<td>5-18, 5-22</td>
</tr>
<tr>
<td>user-defined scalars</td>
<td>5-15, 5-22</td>
</tr>
<tr>
<td>magnetohydrodynamic model</td>
<td>10-1</td>
</tr>
<tr>
<td><strong>Makefile</strong>, 7-2</td>
<td></td>
</tr>
<tr>
<td>mass flow rate</td>
<td>10-2</td>
</tr>
<tr>
<td>mass transfer coefficient UDF, multiphase</td>
<td>4-100</td>
</tr>
<tr>
<td>material property macros</td>
<td>5-15</td>
</tr>
<tr>
<td>material property UDFs for DPM</td>
<td>4-143</td>
</tr>
<tr>
<td>material property UDFs</td>
<td></td>
</tr>
<tr>
<td>for DPM</td>
<td>4-63</td>
</tr>
<tr>
<td>general purpose</td>
<td></td>
</tr>
<tr>
<td>MATERIAL_PROP, 4-136, 4-156</td>
<td></td>
</tr>
<tr>
<td>MATERIAL_PROPERTY, 4-64, 4-67</td>
<td></td>
</tr>
<tr>
<td>MATERIAL_TYPE, 4-156</td>
<td></td>
</tr>
<tr>
<td>Materials panel</td>
<td>8-10, 8-21, 8-23, 8-53, 11-19</td>
</tr>
<tr>
<td>mathematical functions</td>
<td>2-13</td>
</tr>
<tr>
<td><strong>mem.h</strong> header file</td>
<td>5-6, 5-16</td>
</tr>
<tr>
<td>Message</td>
<td>4-131, 6-26, 9-29, 9-36</td>
</tr>
<tr>
<td>message displaying macros, parallel</td>
<td>9-29</td>
</tr>
<tr>
<td>message passing, parallel</td>
<td>9-4</td>
</tr>
<tr>
<td>message displaying macros</td>
<td>9-31</td>
</tr>
<tr>
<td>example</td>
<td></td>
</tr>
<tr>
<td>macros</td>
<td>9-30</td>
</tr>
<tr>
<td><strong>Message0</strong>, 9-29</td>
<td></td>
</tr>
<tr>
<td>metric.h header file</td>
<td>5-3, 5-5, 5-15</td>
</tr>
<tr>
<td>mixing law, thermal conductivity</td>
<td>4-67</td>
</tr>
<tr>
<td>mixture diffusivity</td>
<td>10-3</td>
</tr>
<tr>
<td>mixture domain pointer</td>
<td>6-15</td>
</tr>
</tbody>
</table>
mixture model
  DEFINE macros usage, 1-18
mixture_species_loop, 4-65
model-specific DEFINE macros
  quick reference guide, 4-15
model-specific UDFs
  hooking to FLUENT, 8-8
models.h, 6-27
MOLECON, 4-39, 5-28
momentum source term
  UDF example, 11-12
mp_thread_loop.c, 6-8
mp_thread_loop.f, 6-9
multiphase DEFINE macros
  quick reference guide, 4-97
multiphase flow, getting domain pointer, 6-12
multiphase flows
  user-defined scalar transport equations, 10-7
Multiphase Model panel, 8-32
multiphase UDFs
  cavitation rate, 4-98
data structures, 1-11
DEFINE macros, 1-16
domains, 1-16
Eulerian model, 1-19
exchange property, 4-100
hooking to FLUENT, 8-32
mixture model, 1-18
single-phase applications,
  difference from, 1-16
threads, 1-16
vector exchange property, 4-105
VOF model, 1-17
writing, 1-11
multiphase variables
  macros for accessing, 5-23
MULTIPLE_COMPUTE_NODE_P, 9-17
myid, 9-35
N_DOT, 4-82
N_REQ_UDM, 4-156
N_TIME, 6-23
N_UDM, 6-27
N_UDS, 6-27
ND_DOT, 6-21
ND ND, 4-8, 4-55, 4-56, 4-60, 4-74,
  4-77, 4-78, 4-82, 6-19
ND_SET, 6-20
ND_SUM, 4-8, 4-9, 6-6, 6-19
NNULLP, 4-118, 4-131, 4-148, 6-28
Node, 1-10
node index number, 6-5
node pointer, 5-2
node variable macros, 5-4
node_to_host, 9-16
NODE_X, 4-114
nodes, 1-9
NOx macros, 5-28
NOx Model panel, 8-17
NOx rate UDF, 4-36
NOX_EQN, 5-28
NOX_EQN(NOx), 4-39
NOX_FRATE, 5-28
NOX_FRATE(NOx), 4-39
NOX_RRATE, 5-28
NOX_RRATE(NOx), 4-39
NULLIDX(NOx, SPE), 4-39
NULLP, 4-148, 6-28
number of faces in cell, macro for, 5-4
number of nodes in cell, macro for, 5-4
number of nodes in face, macro for, 5-4
NV CROSS, 4-114
NV_D, 4-88, 4-114
NV DOT, 4-30, 4-118, 4-121
NV_MAG, 4-77, 4-78, 4-103, 4-118, 4-121,
  6-18, 9-9
NV_MAG2, 6-19
NV_S, 4-88, 4-109, 4-114

© Fluent Inc. January 22, 2003
Index

NV_V, 4-82, 4-114, 6-20
NV_V_VS, 6-20
NV_VEC, 4-88, 4-103, 4-109, 4-114, 4-118
NV_VS, 4-131
NV_VS_VS, 6-21
NVVV, 4-114, 6-20

on-demand UDF, 4-10
ONE_COMPUTE_NODE_P, 9-17

P-1 radiation model
implementation, 11-31
P_CELL, 4-136, 4-156
P_CELL_THREAD, 4-136
P_CURRENT_LAW, 4-156, 5-26
P_DEVOL_SPECIES_INDEX, 5-26
P_DIAM, 4-41, 4-44, 4-121, 4-136, 4-140, 5-24
P_DT, 4-148, 4-156, 5-24
P_EVAP_SPECIES_INDEX, 5-26
P_FLOW_RATE, 4-136, 5-24
P_INIT_DIAM, 4-140, 4-154
P_INIT_MASS, 4-41, 4-43, 4-144
P_LF, 5-24
P_MASS, 4-43, 4-126, 4-131, 4-136, 4-140, 4-144, 4-156, 5-24
P_MATERIAL, 4-156
P_NEXT_LAW, 5-26
P_OXID_SPECIES_INDEX, 5-26
P_POS, 5-24
P_PROD_SPECIES_INDEX, 5-26
P_RHO, 4-121, 4-136, 4-140, 5-24
P_T, 4-156, 5-24
P_THREAD, 4-156
P_TIME, 4-126, 5-24
P_USER_REAL, 4-131, 5-26
P_VEL, 4-121, 4-126, 5-24
P_VEL0, 4-121
P_VFF, 5-24

parabolic velocity UDF example, 11-2
PARALLEL, 9-13, 9-39

parallel macros, 9-12
  global logicals, 9-22
  global maximums and
  minimums, 9-21
  global sums, 9-20
  global synchronization, 9-22
parallel UDF example, 9-36
parallel UDFs
  about, 9-1
  communication macros, 9-15
  global reduction macros, 9-18
  macros, 9-12
  overview, 9-1
  predicates, 9-17
  writing files, 9-39
parallelizing your serial UDF, 9-12

particle
  boundary condition UDF, 4-118
  custom law UDF, 4-139
  drag coefficient, 4-128
  erosion and accretion
  rate UDFs, 4-130
  injection initialization UDF, 4-136
  material property UDF, 4-143
  sampling output UDF, 4-141
  scalar update UDF, 4-147
  source term UDF, 4-151
  spray collide UDF, 4-153
  switching law UDF, 4-156

particle diameter, 4-136
particle emissivity, 4-143
particle location, 4-136
particle or droplet diameter, 4-63
particle reaction rate UDF, 4-40
particle scattering factor, 4-143
particle source terms, 4-151
particle surface tension, 4-143
particle velocity, 4-136
particle viscosity, 4-143
partitioned grid, terms, 9-7
partitions, 9-1
phase domain (subdomain) pointer, 6-13
phase domain index, 6-18
Phase Interaction panel, 8-34, 8-36
phase thread (subthread) pointer, 6-14
phase-level threads, 1-11, 6-15
PHASE_DOMAIN_INDEX, 6-8, 6-9, 6-14, 6-18
phase_domain_index, 1-12, 6-14, 6-18
physical property UDF example, 11-18
plasma-enhanced surface reaction modeling, 10-1
pointer array, 6-8, 6-15
pointers, 2-8
  phase domain, 6-14, 6-18
  phase thread, 6-14
postprocessing UDF example
  using user-defined scalars, 11-28
postprocessing UDF, parallel, 9-39
Prandtl number UDFs
  specific dissipation, 4-50
  temperature equation
    diffusion term, 4-51
  thermal wall function, 4-52
  turbulence kinetic energy, 4-47
  turbulent dissipation, 4-46
predicates, parallel UDFs, 9-17
premixed combustion model UDFs
  source term, 4-82
  turbulent flame speed, 4-82
Pressure Inlet panel, 10-9
PRF_CRECV, 9-30
PRF_CRECV_INT, 9-39
PRF_CRECV_REAL, 9-39
PRF_CSEND, 9-30
PRF_CSEND_INT, 9-39
PRF_CSEND_REAL, 9-39
PRF_GIHIGH, 9-18, 9-21
PRF_GILOW, 9-21
PRF_GISUM, 9-18, 9-20
PRF_GLAND, 9-22
PRF_GLOR, 9-22
PRF_GRHIGH, 9-21
PRF_GRLOW, 9-21
PRF_GRSUM, 9-20
PRF_GRSUM1, 9-38
PRF_GSYNC, 9-22
PRINCIPAL_FACE_P, 9-9, 9-26, 9-38
printf, 2-15, 4-7, 4-13, 4-148
PROP_ktc, 4-64
PROP_mu, 4-64
PROP_rho, 4-64
property UDF, 4-63, 4-69
Property_ID, 4-64
radiation scattering phase function, 4-69
radiative transport equation, 4-28
rate of strain, 4-63
reaction rate UDFs
  example, 11-23
  particle, 4-40
  species net mass, 4-34
  surface, 4-76
  volumetric, 4-93
read/write UDF, 4-13
reader, 4-14
reconstruction gradient macros, 5-9
Reynolds Stress Model macros, 5-15
RP macros, 6-22
RP_CELL, 4-136, 4-154, 4-156
RP_Get_Integer, 6-25, 9-38
RP_HOST, 9-13, 9-36, 9-39
RP_NODE, 9-12, 9-13, 9-39
RP_THREAD, 4-154, 4-156
Sample Trajectories panel, 8-52
sampling plane output, 4-141
scalar transport equations, 10-1
scalar update UDF, 4-147
scattering phase UDF, 4-69
Scheme command, 6-24
Scheme interpreter, 6-24
scheme variables, 6-24
Scheme variables, modifying, 6-25
seggregated solver, 1-13
Select File panel, 7-4, 7-7
sg_mem.h header file, 5-17
sg_nox.h header file, 5-28
shared library, 7-2
single phase flows
  user-defined scalar transport
  equations, 10-4
  single-phase applications, 1-16
  slip velocity UDF, 4-105, 4-106
Solid panel, 8-25
solidification, 11-18
solution process
  coupled solver, 1-13
  segregated solver, 1-13
solutions variables and derivatives, 5-2
solver data, access to, 5-1, 5-2
solver variables, 5-2
  accessing, 5-1
source files, 7-1, 7-9
source term UDF
  discrete ordinates model, 4-27
  discrete phase model, 4-151
  for FLUENT transport equations, 4-72, 10-10, 11-12
  for user-defined scalar transport
  equations, 10-6
  premixed combustion model, 4-82
Spalart-Allmaras turbulence model, 4-85
species diffusivity UDF, 4-23
species mass fraction, 4-19, 4-53, 4-76
species net reaction rate UDF, 4-34
specific dissipation Prandtl
  number UDF, 4-50
specific heat, 4-63, 5-2
specular reflectivity UDF, 4-29
spray collide UDF, 4-153
SQR, 4-82, 4-85, 6-29
storage, checking, 6-28
stress condition profile, 4-53
sub_domain_loop, 6-6
sub_thread_loop, 6-8
subdomains, 1-11
subthreads, 1-11
superdomains, 1-11
superthreads, 1-11
Surface Monitors panel, 11-10
surface reaction rate UDF, 4-76
surface tension coefficient, for
  multiphase VOF model, 4-63
switching custom laws for DPM, 4-156
Syamlal drag law, 4-101
T_SAT, 4-104
temperature equation diffusion term
  Prandtl number UDF, 4-51
temperature profile, 4-53
temperature-dependent viscosity, 11-18
thermal conductivity, 4-63
thermal wall function Prandtl
  number UDF, 4-52
Thread, 1-10
  thread pointer, 1-11, 5-2, 6-1
  thread pointer, to array of
    phase threads, 5-2
  thread storage, parallel, 9-10
THREAD_C0, 4-131
THREAD_F_WALL, 4-118
THREAD_ID, 4-77, 4-78, 4-136
thread_loop_c, 4-7, 4-8, 4-131, 6-2
thread_loop_f, 4-131, 6-3
THREAD_MATERIAL, 4-65, 4-67, 4-136, 4-156
THREAD_STORAGE, 4-131, 4-156
THREAD_SUB_THREAD, 4-101, 4-103, 4-104, 4-106, 6-14
THREAD_SUB_THREADS, 6-15
THREAD_SUPER_THREAD, 6-16
THREAD_T0, 4-77, 4-78, 4-88, 4-114, 5-9
THREAD_T1, 4-88, 5-9
THREAD_TYPE, 4-118
threads
  cell, 1-9
  face, 1-9
  fluid, checking, 6-28
  multiphase UDFs, 1-16
  node, 1-9
  phase-level, 1-11, 6-14
  pointers, 6-14, 6-16
  referencing, 1-16
Index

subthreads, 1-11
superthreads, 1-11
variables
and neighboring cell variables, 1-8

Time Step Size, 11-9
time-dependent variables, 6-22
time-stepping control UDF, 4-21
Tracked_Particle, 4-130, 4-139, 4-143, 4-147, 4-151, 4-153, 4-156, 5-24
transient velocity UDF example, 11-7
transport equations
custom source for, 4-72
trigonometric functions, 2-13
tube flow, 11-7
turbine vane, 11-2
turbulence kinetic energy
Prandtl number UDF, 4-47
turbulent dissipation
Prandtl number UDF, 4-46
turbulent dissipation rate profile, 4-53
turbulent kinetic energy profile, 4-53
turbulent premixed combustion
source term, 4-85
turbulent viscosity UDF, 4-85

UDF arguments, 5-2
UDF sample problems, 11-1
udf.h file, 1-5, 7-2
including, 1-5
location of, 7-2

UDFs
arguments, 1-11
arguments not passed, 1-11
compiled, 1-6
compiling, 7-7
data structures, 1-10, 1-16
definition, 1-1
example, step-by-step, 3-1
header files, 1-5
#include statement, 1-5
interpreted, 1-6
interpreting, 7-3
limitations, 1-3
multiphase, 1-16
purpose, 1-3
sequencing, 1-13
single-phase vs. multiphase, 1-16
steps in implementing, 3-1
UDS flux function, 4-87
UDS unsteady function, 4-91
UNIVERSAL_GAS_CONSTANT, 4-41, 4-43, 4-77, 4-78, 4-94, 6-29
UNIX systems, 7-5
unstable simulations, 5-11
Use Contributed CPP, 7-5
user-defined data types, 2-7

User-Defined Function Hooks panel,
8-2–8-4, 8-6, 8-8, 8-12–8-16, 8-18, 8-26, 8-27, 8-31, 8-32, 8-57

User-Defined Functions panel, 8-11, 8-22, 8-24, 8-34, 8-36, 8-54
user-defined mass flux, 10-2

User-Defined Memory panel, 8-7
user-defined memory, 4-10, 4-131, 8-7
user-defined memory variable
example, 5-19
for cells, 5-18
for faces, 5-22

user-defined mixing law UDFs
for density, viscosity, and conductivity
of mixture materials, 4-63

user-defined scalar diffusivity UDF, 4-23
user-defined scalar transport equations
diffusivity UDF, 4-23
examples, 10-3, 11-28
flux term UDF, 4-87
source term UDF, 4-72
unsteady term UDF, 4-91

user-defined scalar transport equations,
maximum number of, 10-7

user-defined scalar variable
example, 5-19
for cells, 5-15
for faces, 5-22

User-Defined Scalars panel, 8-29, 8-30, 10-4, 10-7
user-defined scalars, 10-1, 10-2
  defining, multiphase flow, 10-7
  defining, single phase flow, 10-4
  theory, multiphase flow, 10-2
  theory, single phase flow, 10-1
utilities
  vector, 6-18

vapor pressure, 4-143
vaporization temperature, 4-143
variables not passed as arguments, 6-2
vector dot products, 6-21
vector exchange property UDF, 4-105
vector utilities, 6-18
Velocity Inlet panel, 3-7, 8-20, 10-4, 11-5, 11-8
velocity inlet
  parabolic profile UDF, 11-2
  transient profile UDF, 11-7
velocity profile, 4-53
viscosity property UDF, 4-63, 11-18
Viscous Model panel, 8-19, 8-28
VOF model
  DEFINE macro usage, 1-17
volume fraction, multiphase, 4-53
volume reaction rate UDF, 4-93, 11-23

Wall panel, 8-44
wall heat flux UDF, 4-32
wall impacts, 4-130
wall shear profile, 4-53
Windows NT systems, 7-5
  Microsoft compiler, 7-5
Windows systems, 4-13
working directory, 7-2
writer, 4-14
writing files, parallel, 9-39
writing UDFs
  for multiphase models, 6-2
  grid definitions, 1-8

ZERO COMPUTE NODE P, 9-17
zone ID, 6-16, 6-24
zones
  definition of, 1-8
  ID, 6-16