

Saturn 1 Carte™ C Programming Environment Guide

Document No.: 69266
Revision: Rev 3
Date: 2 February 2016

DirectStream, LLC
4240 N. Nevada Avenue
Colorado Springs, CO 80907
(719) 262-0213
get.directstream.io

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	1 of 241



Copyright © 2016 DirectStream, LLC. All rights reserved.

The contents of this guide are protected under copyright law. This guide, in part or in full, may not be reproduced or transmitted in any form or by any means including but not limited to electronic and hardcopy without written permission from DirectStream, LLC.

DirectStream™, SNAP™, and Carte™ are trademarks of DirectStream, LLC.

MAP® is a registered trademark of DirectStream, LLC.

Intel® is a registered trademark of the Intel Corporation.

Atom™ is a trademark of the Intel Corporation.

Stratix®, Quartus®, and Altera® are registered trademarks of Altera Corporation.

VCS® and Synopsys® are registered trademarks of Synopsys, Inc.

HP® is a registered trademark of the Hewlett-Packard Company.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	2 of 241

6. CODE DEVELOPMENT AND PORTING TO THE MAP PROCESSOR

To take full advantage of the MAP hardware, it is necessary for the user to make modifications to the application. This section describes the required application modifications for MAP execution. These modifications include partitioning the code for MAP execution into new files, restructuring the partitioned code, inserting MAP resource management library calls, inserting calls for Ethernet communications, and inserting calls to the function compiled for MAP execution in the code that executes on the CPU.

6.1 Partitioning the Code

The first step in porting an application to a system utilizing the MAP hardware is to identify some portion of the application such that, when that portion is compiled for the MAP processor, overall performance improves. Loops are often good candidates for execution on the MAP processor. Loop nests and their associated loop bodies, especially those that could be pipelined, have shown execution speed-up. Another potential improvement could be in the process of manipulating single bits from within a long bitstream of data.

Note: A body of code identified for MAP execution must be encapsulated as a standard C function and placed in its own file. This file may contain one function or a MAP function followed by one or more functions to be inlined into the MAP function. The MAP function must be the first function in the file, followed by any functions to be inlined into the MAP function.

Similar to calling traditional microprocessor functions, invoking a MAP processor function involves some computational overhead. First, the user logic must be configured by the microprocessor invoking the MAP processor function. Once the user logic is configured, the microprocessor sends the function parameters to the user logic and subsequently provides a start command to the function (this is roughly analogous to a microprocessor placing a traditional function call on the call stack). Once the MAP processor function finishes execution, the microprocessor is notified of completion and return value sent back to the microprocessor. Depending on the particular implementation of the SNAP interface between the microprocessor and user logic, invoking and returning from MAP processor functions can incur multiple seconds of delay. Ideally, MAP processor functions should execute long enough to amortize this delay.

The function name is the base name for all compiler generated components. Similarly, the filename that contains the function to be compiled, minus the *.mc* suffix, becomes the base name for all compiler generated files. By convention, all files to be compiled with *mcc* are suffixed with *.mc*. The files included as part of inlining may continue to have the *.c* suffix.

Note: Depending on the compilation path, the base name of the file containing the MAP function may be used in the creation of a unique Verilog module identifier. The base name of the *.mc* file should follow the same naming conventions used for C identifiers.

6.1.1 Inlining Functionality

The Carte compiler inlines C functions into a MAP function provided these functions satisfy restrictions described in this section. Inlining allows the user to structure their MAP code into a hierarchy of functions. After the inlining phase has completed, the entire code body is available to the compiler for optimization, allowing optimization across functions and eliminating function call overhead.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	27 of 241

Inlining is invoked in the standard Makefile with additional options to the *MCCFLAGS* and *INLINEDIR* variables. Functions able to be inlined are included in the file containing the MAP function. They may be defined in the same file, following the MAP C function, or as separate files and included using a *#include <filename>* directive. If a C function is identified for inlining but it cannot be inlined, the Carte compiler produces a warning and skips further compilation of the function. Only one MAP function exists after inlining for the main compilation process. If a C function cannot be successfully inlined, compilation halts.

The following is an example of how to take advantage of inlining. The example is found in */opt/src/ci/comp/examples/map_m/C/doc/c-inline*:

```
#include <libmap.h>
/* example.mc */

int64_t foo1(int64_t in_foo1);
int64_t foo2(int64_t in_foo2);

void MAP_foo(int64_t my_arg, int64_t *my_foo, int mapnum) {
    *my_foo= foo1(my_arg);
}

int64_t foo1(int64_t infoo)
{
    int64_t your_foo = 6;
    int64_t tmp_foo;
    tmp_foo = foo2(your_foo);
    return(tmp_foo+infoo);
}

int64_t foo2(int64_t x)
{
    return (x + 1);
}
```

Without inlining, all code executing on the MAP processor during a call to a MAP function must reside in the body of the MAP function, in this case *MAP_foo*. No external calls are allowed, except to user or DirectStream-defined macros. With inlining, it is possible for the MAP function to call other functions, which are automatically inlined into the MAP function. Inlining makes it easy to retain the code structure present in the code selected for MAP execution.

If the file *example.mc* contains the MAP function *MAP_foo* and all the functions that *MAP_foo* calls are inlined, *MAP_foo* must be the first function present in the file. Function prototypes for the inlined functions must precede the definition of the MAP function. This particular example shows the inlining capability of inlining a function, that itself contains a function to be inlined.

To use inlining, modify the standard Makefile template by adding the *-inline* option to the *MCCFLAGS* Makefile variable. For the example, *MCCFLAGS* is set as follows:

```
MCCFLAGS          = -inline=name:foo1,foo2,levels:2
```

The *-inline* option has two optional clauses: *name:* and *levels:*.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	28 of 241

The *name:* clause specifies a list of function names to be inlined. If *name:* is specified, a list of the function names to be inlined is listed following the *name:* keyword. If the *name:* clause is not specified, the compiler attempts to inline all the functions included in the file with the MAP function. If this attempt is not successful, a compilation message is issued and compilation halts.

The *levels:* clause specifies the number of levels of functions in the call graph to inline. The format of the *levels:* clause is *levels:n*, where *n* is the number of levels to inline. If a *levels:* clause is not specified, one level is the default and only functions called from the MAP function are inlined. In the example, if *levels* was not specified, *foo1* is inlined into *MAP_foo*, but *foo2* was not brought into *foo1* first, so it remains an external call. This results in a compilation error, as the MAP function cannot call external functions except to user or DirectStream-defined HDL macros.

When this example is successfully compiled, the following compilation messages are issued:

```
Inizio/MAP Linux Release 5.0 (0.0)
Copyright 2002, SRC Computers,LLC All Rights Reserved.
Inizio/MAP Linux Release 5.0 (0.0)
Copyright 2002, SRC Computers,LLC All Rights Reserved.
function foo1 inlined at line 12
function foo2 inlined at line 5
Inizio/MAP Linux Release 5.0 (0.0)
Copyright 2002, SRC Computers,LLC All Rights Reserved.
function foo1 inlined at line 12
function foo2 inlined at line 5
```

This indicates the inlining was successful.

Functions to be inlined may be contained in a separate file from the MAP file and included in the MAP compilation using the *#include* directive. The functions are included by the pre-processor at compilation time and inlined in the same fashion as described above. The *#include* directives must appear following the MAP function to satisfy the requirement that the MAP function be the first function in the file. Functions inlined in this way cannot be compiled in debug mode if they contain DirectStream-defined macros. For successful debug execution, the Carte compiler modifies these macros.

Note: External files with inlining functions that contain DirectStream-defined macros prevent the Carte compiler from generating modified versions of those functions for debug mode. Only functions present in the file containing the MAP function are modified for debug mode. If any inlined functions contain DirectStream-defined macros, place these functions in the same file as the MAP function.

Functions that are candidates for inlining in the MAP function must meet the same restrictions as the MAP function, with some additional restrictions:

- The function cannot have a return type of *struct* or be a function with a *struct* argument. MAP functions themselves are only type *void*.
- The function cannot contain a switch statement.
- The function cannot reference a static variable whose definition is nested within the function. MAP functions cannot have any static data or reference global file-scope variables.
- The function must reside in a file with a *.c* suffix or be appended to the C file that contains the MAP function. The MAP function must still be the first function presented in the C file to be compiled.
- Stream variables may not be used as arguments to inlined functions.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	29 of 241

- The function may not contain any DirectStream-defined macro using streams, user-defined macro using streams, or any macros that have stateful as a macro attribute.
- The function may contain initialized data using an array declaration with the *const* type qualifier specified. Each invocation of the inlined function has its own copy of these initialized arrays after inlining.
- The function cannot contain pragmas to identify parallel constructs.
- The function may have pragmas that operate on loop bodies inside the function, but those pragmas are not recognized after the function is inlined into the MAP C function.

6.1.2 Data Partitioning

The MAP function contains the computational portion of code to be executed on the MAP processor. This function must be modified to manage the data movement to and from the MAP processor's OBM or its VLM. Data movement function calls must be inserted to manage data movement between System Common Memory (SCM) and OBM. At the beginning of program execution, all data resides in SCM on the CPU or input from an external source, such as Ethernet. Array data needed by a MAP function is only transferred from SCM to the MAP processor using Direct Memory Access (DMA) operations. The exception is statically initialized constant array data that reside in the MAP processor's RAM blocks. The DMA operations move data into and out of OBM, or they stream data from SCM and/or VLM directly into the UL chip. Results that reside on OBM are persistent for later use by MAP functions, or moved back to SCM or VLM via another DMA operation. Results that reside on VLM are not guaranteed to persist across calls to MAP functions. The C compiler generates the code to automatically move scalar formal parameters of the MAP function to and from the MAP processor as needed.

6.2 Restrictions on the Code

The computational portion of the code may need to be further modified to meet the restrictions of the Carte compiler. The majority of these restrictions pertain to loops that could be pipelined and those functions that are candidates for inlining.

Note: A MAP function must be a function returning void. Any result values must be returned through the formal parameter list. A MAP function must have a return type *void*. Functions returning types other than *void* cannot be called directly from the CPU. These functions may be used as inlining candidates.

6.2.1 Formal Parameters to a MAP Function

Global data (*externs*) may not be referenced or defined by the MAP function with the exception of the specifically named structures associated with OBM. Data referenced or defined by the MAP function and also referenced or defined by the calling function must be passed via formal parameters to the MAP function or transferred between the MAP function and the calling function by a DMA operation.

Array parameters must be declared with the square bracket notation [] rather than as pointers to data. Pointer arithmetic is not supported in MAP functions.

The last formal parameter in the MAP function's argument list is a *mapid* that must be of type *int*. The *mapid* is the number that indicates the MAP processor on which the function is to execute. The MAP processors allocated to a job are identified by an integer in the range of 0 through *n-1*, where *n* is the number of MAP processors allocated to the job. Refer to Section 11.2 for more information about MAP allocation.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	30 of 241

6.2.2 External Function References in the MAP Function

The MAP function must not contain any external function references except those linked to either DirectStream-defined or user-defined HDL macros (refer to Section 19) or replaced by inlining code. In this sense, after inlining has completed, functions are wholly contained on the MAP processor.

No additional system header files besides the *libmap.h* and *carte_socket.h* header file are allowed in the MAP function. No system calls or other runtime functions requiring OS intervention are allowed. An example is memory allocation. An exception to this restriction, *printf* statements may be inserted to help with running the MAP function in debug mode. These statements are treated as comments in other compilation modes. Assertions are not allowed within MAP functions as they alter the generated source code during compilation and depend on runtime support. Define the C pre-processor macro *NDEBUG* to disable assertions inside the MAP function.

Macros called from within the MAP function cannot have the same name as the MAP function itself. Such a reference is considered a recursive call to the MAP function. The Carte compiler does not support recursion.

6.2.3 Intrinsics and Data Types

The signed and unsigned data types *char*, *short*, *int*, *long*, and *long long* are supported. For consistency, these data types map to the same number of bits as on the Intel processor. An *int* variable represents a 32-bit integer in the MAP function. In accordance with the C standard, *char* and *short* operands are promoted to *int* prior to performing any operations. The compiler performs the implicit data conversions as needed.

The intrinsic operators *+*, *-*, ***, */*, *%*, *==*, *!=*, *>*, *>=*, *<*, and *<=* are fully supported for these types. The bitwise operators *&*, *|*, and *!* are provided for signed and unsigned *int* and *long long* (32- and 64-bit) data types. Left and right shift operators, *<<* and *>>*, are supported for signed and unsigned *int* and *long long* types. Logical operators *&&*, *!*, and *||* are supported.

The following intrinsic mathematical functions are available for 32-bit integer data: *fabsf*, *sqrtf*, *expf*, *logf*, *log10f*, *log2f*, *powf*, *cosf*, *sinf*, *tanf*, *acosf*, *asinf*, *atanf*, *atan2f*, *coshf*, *sinhf*, and *tanhf*. The *sqrt* and *fabs* intrinsics are supported for 64-bit integer inputs. No additional header files are needed in the MAP function to define these intrinsic functions.

6.2.4 Declaring and Referencing Data

Data must satisfy this list of criteria in a MAP function:

- Array formal parameters must not be referenced or defined except as parameters to data movement functions.
- Array formal parameters must be declared using square brackets [] to distinguish them from formal parameters that are pointers to scalars.
- Array formal parameters must be of type *long long*, unsigned *long long*, or *double*.
- There may be a maximum of 64 reads and 16 writes to a local array in RAM Blocks across the entire MAP function (as considered after inlining). The distribution of the reads and writes among different code blocks does not matter. See Section 6.3.2 for more details about arrays in RAM Blocks.
- See Section 6.3.2 for more details about arrays in OBM.
- Only static data declared as an array with the *const* type qualifier may be initialized and then referenced. Other static data declarations are not supported, including globally referenced data.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	31 of 241

6.3 Modifications to the MAP Function

Once the body of code is identified for MAP execution, the MAP function body is constructed. All data shared between the MAP function and the calling function must be passed as formal parameters to the MAP function or transferred between the MAP function and the calling function by DMA operations. This includes an argument defining the MAP processor number. The runtime library uses this argument to identify the logical MAP processor executing the function.

Note: The *map_id* argument is the last argument in the argument list.

6.3.1 *libmap.h* Include File

The header file *libmap.h* contains system-defined symbolically named constants and function prototypes for the MAP library functions discussed in this document. Include this header file in both MAP functions and functions that call MAP functions. Some of the symbolic constants defined in *libmap.h* are described in Table 6.

Table 6 *libmap.h* Include File

Symbolic Constant	Description
<i>MAX_OBM_SIZE</i>	Maximum number of 64-bit words accessible in an OBM bank
<i>CM2OBM</i>	Directional indicator for DMA transfer from CPU memory
<i>OBM2CM</i>	Directional indicator for DMA transfer to CPU memory
<i>VLM_BANK [0,1]</i>	VLM bank designator
<i>STREAM_TO_PORT</i>	Directional indicator for stream connection to a DMA
<i>PORT_TO_STREAM</i>	Directional indicator for stream connection to a DMA
<i>map_m</i>	MAP board designator
<i>PATH_0</i>	DMA data path designator – ignored for <i>map_m</i>
<i>PATH_1</i>	DMA data path designator – ignored for <i>map_m</i>

6.3.2 Memory Allocation

Memory that is directly accessed must be allocated at compile time on the MAP processor as C arrays using square bracket notation []. These arrays load and store any data while being computed on the MAP processor. Depending on how the arrays are declared, all arrays reside in one of two places, RAM Blocks or OBM banks.

On the Saturn 1 server cartridge, there is an additional memory (VLM) that is only accessed using DirectStream-defined functions. There are no special declarations provided to allocate space in the VLM banks.

6.3.2.1 Local Arrays in RAM Blocks

The Carte compiler supports locally declared arrays of one and two dimensions by allocating them in RAM Blocks within the UL. The number of RAM Block arrays in a MAP function is limited by the total memory in the chip, size of the arrays, and data types of the arrays. A RAM Block may contain only one array. Two small arrays cannot share a RAM Block. Larger arrays may occupy more than one RAM Block.

Note: The UL on the Saturn 1 server cartridge supports array sizes of up to sixty-five thousand five hundred thirty-six 64-bit elements, one hundred thirty-one thousand seventy-two 32-bit elements, two hundred sixty-two thousand one hundred forty-four 16-bit elements, and five hundred twenty-four thousand two hundred eighty-eight 8-bit elements.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	32 of 241

The RAM Blocks are dual-ported. The Carte compiler uses one port for reading and one port for writing. A read and a write can happen concurrently in the RAM Blocks. See Section 17.1 for a discussion of load/store scheduling and data dependence issues.

6.3.2.2 Constant Arrays in RAM Blocks

The Carte compiler supports constant RAM Block arrays with static initializers. For constant arrays, both of the array's ports are used for reading so two reads can occur concurrently:

```
const int8_t T[512] = {
    23, 55, 73, 98, 34, 11, 26, 90,
    ...
    41, 27, 86, 54, 35, 82, 36, 77
};
...

for (i=0; i<n; i++) {
    idx0 = ...;
    idx1 = ...;
    v = T[idx0] + T[idx1];
    ...
}
```

The compiler uses 1 RAM Block unit for array *T* and is initialized with the 512 values specified. In the loop, the 2 reads of *T* happen concurrently so the loop runs at full speed.

Loops with more than two accesses to a *const* array have their accesses distributed between the two ports of the array. For example, a loop with six accesses to *T* requires three clocks per iteration. Note that the *const* keyword must be included in the array declaration to use this feature.

Note: The size of constant arrays in RAM Blocks is limited to six thousand one hundred forty-four 64-bit elements, twelve thousand two hundred eighty-eight 32-bit elements, twenty-four thousand five hundred seventy-six 16-bit elements, and thirty-two thousand seven hundred sixty-eight 8-bit elements.

6.3.2.3 Arrays in OBM Banks

The 4 OBM banks are called *A*, *B*, *C*, and *D*. Each contains *MAX_OBM_SIZE* 64-bit words of storage and each may be accessed (read or write) on every clock tick.

To declare OBM memory at the top of the MAP function, use DirectStream-defined macro syntax.

```
OBM_BANK_<x> (<array name>, <element type>, <extent>)
OBM_BANK_<x>_2D (<array name>, <element type>, <extent 0>, <extent 1>)
OBM_BANK_<x>_2_ARRAYS (<array name 0>, <element type 0>, <extent 0>,
    <array name 1>, <element type 1>, <extent 1>)
OBM_BANK_<x>_3_ARRAYS (<array name 0>, <element type 0>, <extent 0>,
    <array name 1>, <element type 1>, <extent 1>
    <array name 2>, <element type 2>, <extent 2>)
```

In these declarations, *<x>* is an OBM bank, *A* through *D*; *<array name>* is the array name being declared; *<element type>* is the array type; and *<extent>* is the number of array elements in a dimension.

Note: *<element type>* must be a 64-bit type such as *int64_t*, *uint64_t*, or *double*.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	33 of 241

For example:

```
OBM_BANK_A (nodes, int64_t, 1000)
OBM_BANK_A_2D (pressure, int64_t, 100, 200)
OBM_BANK_A_2_ARRAYS (flux_x, int64_t, 500, flux_y, int64_t, 800)
```

The first declares an array of one thousand 64-bit integers in bank *A* called *nodes*. The second declares a 100 X 200 array in bank *B* called *pressure*. The third declares two arrays in bank *C*: *flux_x* is an array of five hundred 64-bit integers and *flux_y* is an array of eight hundred 64-bit integers.

6.3.2.4 Memory in VLM

There are two independent banks of 16 GB SDRAM referred to as VLM. VLM banks are accessed (read/write) by invoking DirectStream-defined functions that can transfer four 64-bit words per clock cycle. More information on VLM functions is found in Section 9.3.

6.3.3 Exploiting Parallelism

In a typical execution of the MAP hardware generated by the Carte compiler, one code block is active at a time. A code block is viewed as a set of sequential hardware instructions executed serially without any branches or exits out of that block. The Carte compiler identifies and exploits parallelism within each code block, especially within a pipelined loop. Performance may benefit further when different sections of the original code execute in parallel independently. This subsection describes how to exploit parallel execution of different code blocks.

A parallel region contains multiple parallel sections as illustrated in Figure 4. When normal execution reaches a parallel region, a *FORK* code block is initiated and execution simultaneously starts at the beginning of all of the parallel sections. There is an active code block in each of the parallel sections until that parallel section completes (reaches the special *JOIN* code block). Only when all parallel sections are complete does execution continue from the *JOIN* code block.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	34 of 241

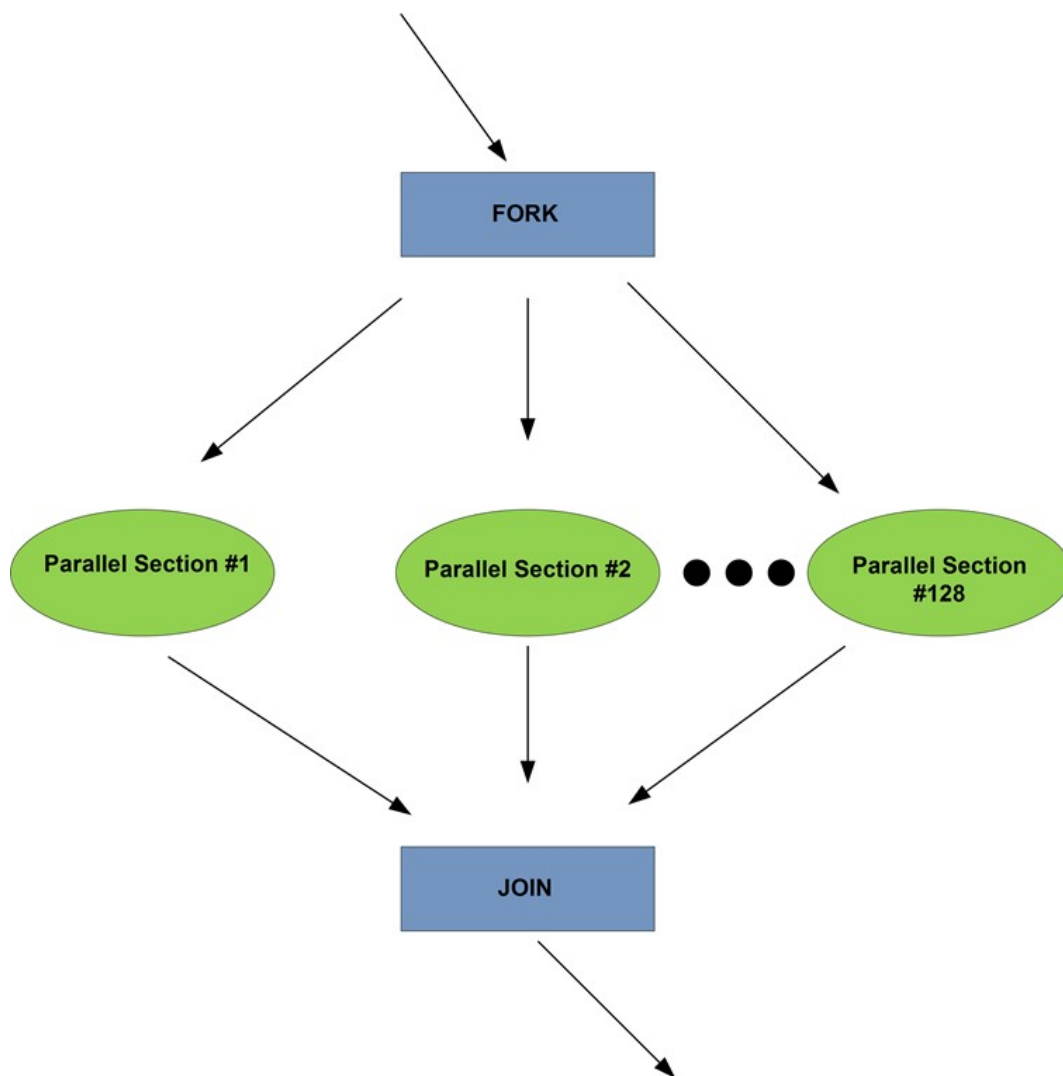


Figure 4 Parallel Region

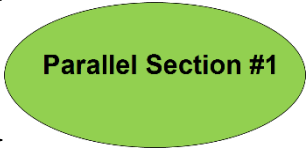
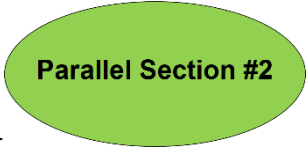
Each parallel section may contain any arbitrary structured control flow including if-then-else statements, loops, and pipelined loops. External macros with variable latency are allowed. Nesting of parallel regions is not permitted. Each parallel region may contain up to 128 parallel sections. There is no pre-defined limit on the number of parallel regions in a program.

The parallel syntax is similar to the Open Multi-Processing (OpenMP) parallel syntax. This allows substitution of the *src* pragmas, indicating parallel regions and sections with OpenMP equivalents, so that parallel execution is enabled in debug compilation.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	35 of 241

The general syntax indicating parallel sections in a parallel region is shown below:

```
void ex4 (int aa, int bb, int *cc, int mapnn)
{
  int i;

  #pragma src parallel sections
  {
    #pragma src section
    {
      
    }
    #pragma src section
    {
      
    }
    ...
  }
}
```

The *#pragma* is standard C syntax and causes this statement to be ignored on other hardware platforms. Immediately following the *#pragma* is the *src* delimiter. This is detected by the Carte compiler, which parses the statement. The corresponding OpenMP delimiter in the C debug code generated during debug compilation mode replaces this delimiter. The following subset of the OpenMP keywords is supported: *parallel sections* and *section*.

A concrete example of a parallel region is shown in the following code example:

```
void ex4 (int aa, int bb, int *cc, int mapnn)
{
  int sum1, sum3;

  #pragma src parallel sections
  {
    #pragma src section
    {
      sum1 = aa - bb;
    }
    #pragma src section
    {
      int sum2;          // temp var
      sum2 = aa + bb;
      sum3 = sum2 + aa;
    }
  }
  *cc = sum1 ^ sum3;
}
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	36 of 241

6.3.3.1 Using Independent Parallel Sections

The simplest use of parallel sections is when they are performing completely independent work (i.e., not accessing any shared data or otherwise communicating with each other).

There are two different mechanisms used to ensure that the scalar variables in the different parallel sections are not shared. The first and obvious approach is to give unique names to the variables. The second approach is to use C's scoping rules as illustrated below. This allows the use of multiple variables that are distinct even though they have the same name. This mechanism works with RAM Block arrays but not with OBM arrays:

```
#pragma src parallel sections
{
    #pragma src section
    {
        int i, a[N];
        for (i=0; i<N; i++) {
            a[i] = i;
            ....
        }
    }
    #pragma src section
    {
        int i, a[N];
        for (i=0; i<N; i++) {
            a[i] = i<<2;
            ....
        }
    }
}
```

6.3.3.2 Using Cooperating Parallel Sections

It may be useful for parallel sections to access shared data in some situations. Examples are transferring data from producer to consumer or accessing a shared work queue. This section outlines some guidelines and restrictions for sharing access to scalars, RAM Block arrays, and OBM arrays. The details are different in each case.

Note: The compiler warns, but does not prevent accesses to the same RAM Blocks/OBM bank from parallel code blocks. Accesses to the same scalar variable from different parallel code blocks do not generate a warning.

The following considerations apply when multiple sections in the same region access the same RAM Blocks or OBM arrays:

- OBM reads and writes to a bank share single port. The compiler's hardware scheduling considers only one parallel section at a time and prevents conflict only within a section. The scheduler cannot prevent conflicts between different parallel sections. The programmer is responsible for ensuring that only one section is accessing a bank at a time. It is not safe for multiple sections to be accessing the same bank simultaneously even when all of the accesses are loads. Nor is it safe even if they are guaranteed to be accessing different elements of the array. In both cases, the multiple addresses sent to the OBM port interfere with each other, leading to unpredictable results. It is important to note that loads within a pipelined loop always execute every iteration even if they are contained in a conditional statement such as the if-then-else clause.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	37 of 241

- RAM Block restrictions are similar except that one section may be reading a RAM Block bank while another section may be writing the same RAM Block bank at the same time because a RAM Block has separate read and write ports.

The following considerations apply when the programmer wishes a scalar value written in one section to be visible in another section:

- The same scalar variable must be visible to the different sections. It must be declared in a scope that contains the parallel region.
- A scalar variable is read by a code block only at the code block's start of execution and written only at the end of its execution. During execution, the code block reads and writes a local copy of the variable. A scalar written by code block *A* is not visible to code block *B* in another parallel section if their execution overlaps. Code block *A* must finish before code block *B* starts if code block *B* is to see the value written by code block *A*.
- Pipelined loops read each scalar once at the beginning of loop execution and then when circulating a local copy. Once a pipelined loop is executing, it cannot see any changes made to the variable by other parallel sections. For a loop to read an updated version of the variable in each iteration (as desired in a *busy wait* loop), the programmer must force a non-pipelined implementation of the loop. This is accomplished by adding the line `#pragma loop noswpipe` immediately before the loop. With a non-pipelined loop, the value is reread from its global (shared) location each iteration.
- The situation with scalar writes is analogous. A pipelined loop writes back its scalars only on exit. In contrast, a non-pipelined loop writes back its scalars at the end of every iteration.
- The programmer needs to warn the compiler that the variable may change in unexpected ways. Accomplish this by adding the *volatile* attribute to the variable's declaration. Without this knowledge, the compiler may make local optimizations that are incorrect in situations where another section is changing the variable's value.

The following code example illustrates the use of parallel sections. This example is found in `/opt/srcci/comp/examples/map_m/C/doc/c-parallel`. The following points are noted:

- The producer and consumer each handle a large chunk of data at a time. Handshaking is performed once per chunk. It consumes a small amount of overall processing time. The elements in each chunk are processed in a pipelined loop to achieve high throughput.
- Each handshaking *while* loop must be non-pipelined so it sees updated versions of the shared *tsent* and *trcvd* variables in each iteration.
- *tsent* and *trcvd* are declared volatile so the compiler does not optimize them as *loop invariant* in the handshaking loops.
- The *pstalls* and *cstalls* variables keep track of the number of operations spent in the handshaking loops by the producer and consumer sections respectively. They are not needed for correct operation.

```
#include <libmap.h>
#define BUFFER_SIZE 2048
#define BLOCK_SIZE 512
#define COUNT 8192

void ex4 (int *val_in, int *pstallo, int *cstallo, int mapnn)
{
    int sum;
    int fifo[BUFFER_SIZE];
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	38 of 241

```
volatile int tsent, trcvd;
int pstall, cstall;

sum = 0;
tsent = trcvd = 0;
cstall = pstall = 0;

#pragma src parallel section
{
    ///--- "producer"
    #pragma src section
    {
        int j, ind, data;
        for (tsent=0; tsent<COUNT; ) {

            ///--- handshake: stall if fifo is full
            #pragma loop noswpipe
            while ( (tsent - trcvd) > (BUFFER_SIZE - BLOCK_SIZE) )
                pstall++;

            ///--- produce a chunk of data
            ind = tsent % BUFFER_SIZE;
            data = tsent;
            for (j=0; j<BLOCK_SIZE; j++) {
                int a;
                a = data*data;
                data++;
                fifo[ind++] = a;
            }
            tsent += BLOCK_SIZE;
        }
    }
    ///--- "consumer"
    #pragma src section
    {
        int j, ind;
        for (trcvd = 0; trcvd<COUNT; ) {
            ///--- handshake: stall if fifo is empty
            #pragma loop noswpipe
            while (tsent == trcvd)
                cstall++;
            ///--- receive a chunk
            ind = trcvd % BUFFER_SIZE;
            for (j=0; j<BLOCK_SIZE; j++) {
                int a;
                a = fifo[ind++];
                sum ^= a*a;
            }
            trcvd += BLOCK_SIZE;
        }
    }
}
*val_in = sum;
*pstallo = pstall;
*cstallo = cstall;
}
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	39 of 241

7. STREAMS

A stream is a DirectStream-defined data structure that allows a flexible way to communicate between concurrent loops that produce data values and loops that consume those data values. Users may declare streams as local variables in a MAP function and reference them in various ways in the code.

7.1 Stream Buffers

Since stream data values are produced from one loop inside of a MAP C function and then consumed from a different loop, another data structure provides a buffer so that stream data values are produced at one rate while potentially being consumed at another rate.

Figure 5 shows a stream buffer connecting a producer stream and a consumer stream together. There is a data path (of 8-, 16-, 32-, 64-, 128-, 192-, 256-, 320-, 512-, or 1024-bit width), a valid signal that travels with the data, and a stall signal that travels back to the producer. The stream buffer eliminates the need for the producer stream and consumer stream to be tightly synchronized. The consumer stream takes data values when they are ready and at the rate the loop generating the consumer stream runs. The producer stream puts data values into the buffer, but stalls if the buffer is approaching full. Only the data path is visible in the MAP code. The other signals are created and connected automatically by the compiler.

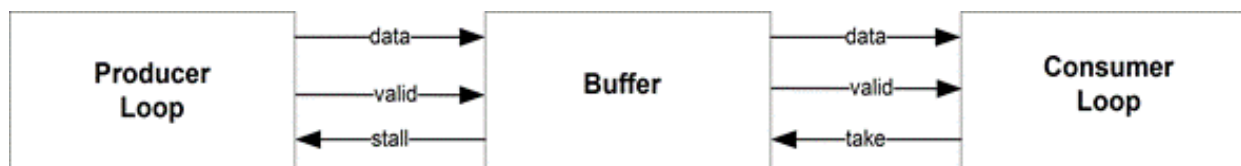


Figure 5 Internal Buffer and Signals Used by a Stream

7.2 Stream Declaration and Basic Functions

Consider the following example:

```

Stream_64 S0;    // declaration of stream
#pragma src parallel sections
{
    #pragma src section
    {
        int i;
        for (i=0; i<sz; i++)
            put_stream_64 (&S0, AL[i]+42, 1);
    }

    #pragma src section
    {
        int i;
        for (i=0; i<sz; i++)
            get_stream_64 (&S0, &BL[i]);
    }
}

```

The stream *S0* forms a connection between the first loop, a producer of data values, and the second loop, a consumer of those data values. The two loops run concurrently and the iterations of the consumer loop are triggered by the presence of data values from the producer loop.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	45 of 241

Streams are declared using DirectStream-defined typedefs. The declaration is:

```
Stream_n <stream>
```

Where n is the stream bitwidth (8, 16, 32, 64, 128, 256, 192, 320, 512, or 1024).

All stream variables must be declared this way:

```
put_stream_<n>
```

```
get_stream_<n>
```

Where n is a stream bitwidth (8, 16, 32, 64, 128, 192, 256, 320, 512, or 1024).

Stream sizes greater than 64-bits are referred to as *wide streams*, as they are wider than any scalar data type in C. The first parameter of each put and get function is a reference to a declared stream variable. A data value and an enable are the remaining parameters of the *put_stream_<n>* function. The second parameter of the *get_stream_<n>* function is a reference to a data variable that receives the fetched value from the stream. For wide streams, there are multiple 64-bit data values in the function argument list corresponding to 64-bit chunks of the wide stream value.

The prototypes of get and put functions are:

```
void put_stream_8   (Stream_8*, int8_t, int);
void put_stream_16  (Stream_16*, int16_t, int);
void put_stream_32  (Stream_32*, int32_t, int);
void put_stream_64  (Stream_64*, int64_t, int);
void put_stream_128 (Stream_128*, int64_t, int64_t, int);
void put_stream_192 (Stream_192*, int64_t, int64_t, int64_t, int);
void put_stream_256 (Stream_256*, int64_t, int64_t, int64_t, int64_t, int);
void put_stream_320 (Stream_320*, int64_t, int64_t, int64_t, int64_t, int64_t, int);
void put_stream_512 (Stream_512*, int64_t, int64_t, int64_t, int64_t, int64_t, int64_t,
                        int64_t, int64_t, int);
void put_stream_1024 (Stream_1024*, int64_t, int64_t, int64_t, int64_t, int64_t, int64_t,
                        int64_t, int64_t, int64_t, int64_t, int64_t, int64_t,
                        int64_t, int64_t, int64_t, int);
void get_stream_8   (Stream_8*, int8_t*);
void get_stream_16  (Stream_16*, int16_t*);
void get_stream_32  (Stream_32*, int32_t*);
void get_stream_64  (Stream_64*, int64_t*);
void get_stream_128 (Stream_128*, int64_t*, int64_t*);
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	46 of 241

```
void get_stream_192 (Stream_192*, int64_t*, int64_t*, int64_t*);

void get_stream_256 (Stream_256*, int64_t*, int64_t*, int64_t*, int64_t*);

void get_stream_320 (Stream_320*, int64_t*, int64_t*, int64_t*, int64_t*, int64_t*);

void get_stream_512 (Stream_512*, int64_t*, int64_t*, int64_t*, int64_t*, int64_t*, int64_t*,
                    int64_t*, int64_t*);

void get_stream_1024 (Stream_1024*, int64_t*, int64_t*, int64_t*, int64_t*, int64_t*,
                    int64_t*, int64_t*, int64_t*, int64_t*, int64_t*, int64_t*,
                    int64_t*, int64_t*, int64_t*, int64_t*);
```

There are functions to handle float and double values. Float values use the *Stream_32* typedef declaration and double data values use the *Stream_64* typedef declaration:

```
void put_stream_flt_32 (Stream_32*, float, int);

void put_stream_dbl_64 (Stream_64*, double, int);

void put_stream_dbl_128 (Stream_128*, double, double, int);

void put_stream_dbl_192 (Stream_192*, double, double, double, int);

void put_stream_dbl_256 (Stream_256*, double, double, double, double, int);

void put_stream_dbl_320 (Stream_320*, double, double, double, double, double, int);

void put_stream_dbl_512 (Stream_512*, double, double, double, double, double, double, double,
                    double, int);

void put_stream_dbl_1024 (Stream_1024*, double, double, double, double, double, double, double,
                    double, double, double, double, double, double, double, double,
                    double, double, int);

void get_stream_flt_32 (Stream_32*, float*);

void get_stream_dbl_64 (Stream_64*, double*);

void get_stream_dbl_128 (Stream_128*, double*, double*);

void get_stream_dbl_192 (Stream_192*, double*, double*, double*);

void get_stream_dbl_256 (Stream_256*, double*, double*, double*, double*);

void get_stream_dbl_320 (Stream_320*, double*, double*, double*, double*, double*);

void get_stream_dbl_512 (Stream_512*, double*, double*, double*, double*, double*, double*,
                    double*, double*);

void get_stream_dbl_1024 (Stream_1024*, double*, double*, double*, double*, double*, double*,
                    double*, double*, double*, double*, double*, double*, double*, double*,
                    double*, double*, double*);
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	47 of 241

The enable parameter (the last parameter in each *put_stream_<n>* function) makes it possible for an iteration of the producer loop to skip sending a data value to the producer stream, based on testing it for true or false.

7.3 Stream Loop Behavior

Multiple streams of varying widths may be declared in a MAP C function. Loops may reference them as producers, consumers, or both. At compilation, the Carte compiler uses the stream parameter to physically connect one producer to one consumer. It is not possible to declare or dynamically manipulate stream pointers in a MAP code. Every stream has exactly one producer and one consumer.

It is natural to view a set of stream loops and operators as a network of processes connected by streams. Figure 6 and the accompanying code example show five loops communicating using five streams. If all of the loops are able to run at full speed (i.e., one clock per iteration), the whole network processes a value in every stream on every clock. This makes the efficiency of the network virtually identical to the case where all the loops were fused into one loop.

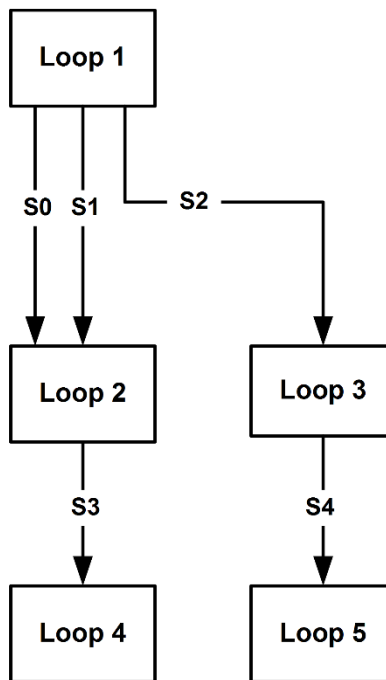


Figure 6 Example of Multiple Loops that Interact with Stream

```

#pragma src parallel sections
{
    #pragma src section
    { // Loop 1
        int i;
        for (i=0; i<num; i++) {
            put_stream_64 (&S0, AL[i]+42, 1);
            put_stream_64 (&S1, BL[i]+17, 1);
            put_stream_64 (&S2, AL[i]+BL[i], 1);
        }
    }
}

```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	48 of 241

```
#pragma src section
{ // Loop 2
    int64_t va, vb;
    int i;
    for (i=0; i<num; i++) {
        get_stream_64 (&S0, &va);
        get_stream_64 (&S1, &vb);
        put_stream_64 (&S3, va+vb, 1);
    }
}

#pragma src section
{ // Loop 3
    int64_t va;
    int i;
    for (i=0; i<num; i++) {
        get_stream_64 (&S2, &va);
        put_stream_64 (&S4, va+10, 1);
    }
}

#pragma src section
{ // Loop 4
    int i;
    for (i=0; i<num; i++) {
        get_stream_64 (&S3, &CL[i]);
    }
}

#pragma src section
{ // Loop 5
    int i;
    for (i=0; i<num; i++) {
        get_stream_64 (&S4, &DL[i]);
    }
}
}
```

Streams allow flexibility in the case where loops run at different speeds. For example, if data dependencies in Loop 1 force it to take two clocks per iteration, the other loops pace themselves accordingly. If Loop 5 runs slowly, its flow control stalls Loop 3 from time-to-time. That in turn causes Loop 1 to stall and the effect in throughput is felt in Loop 2 and Loop 4. Regardless, the network computes the correct values in all cases.

7.4 Stream Coding Restrictions

There are a number of restrictions in the use of streams:

1. Each stream must have exactly one producer.
2. Each stream must have exactly one consumer.
3. The network graph described by the loops of a parallel region and the streams that connect them must be acyclic.
4. All of the parallel sections in a parallel region must terminate before the region terminates. This means that all of the stream producer and consumer loops must terminate.
5. A stream may not be both produced and consumed by the same loop.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	49 of 241

6. Functions to *get_stream* and *put_stream* may not be inside of conditionals.
7. Streams cannot be used in functions to be inlined into the MAP code.
8. No more than 1024 streams are allowed in a MAP function.

The following are not allowed:

- Multiple loops feeding the same stream (restriction 1).
- Reuse of a stream by different parallel regions (restriction 2).

7.5 Runtime Behavior of Streams

When an executing MAP function reaches a parallel region, all stream buffers associated with the streams of that parallel region are cleared. This eliminates any stray data values left by a previous execution of that region. All of the parallel code sections of that parallel region are then started. Loops that do not consume streams iterate immediately. Loops that consume one or more streams have their iterations triggered by the presence of data values in their associated stream buffers. The first data value written to each buffer is available to its consumer in the clock tick immediately following the write of that data value into the buffer.

As long as producer and consumer loops are running at full speed (i.e., one iteration per clock), full throughput is achieved. This means there are no bubbles or gaps of data values in the loop pipelines. A consumer loop stalls if any of its input stream buffers are empty. A producer loop stalls if any of the stream buffers it targets is approaching full.

In general, MAP C functions utilizing streams are written such that an equal number of data values are produced and consumed for a given stream. This equality rule may be bent slightly, though it is difficult for the writer of the MAP C functions to know what is acceptable. A producer stream potentially produces more data values than its consumer stream takes, as long as the excess data values are able to fit into the stream buffer that feeds the consumer stream. A safe size to overrun data values into the stream buffer is not known because the compiler determines the size of the stream buffer, not the person coding the MAP C function. If the number of data values that are overrun exceeds the buffer size, execution of the MAP C function deadlocks.

7.6 Stream Buffer Sizes

The buffers that interconnect stream producers and consumers have their sizes determined by the compiler. Because of the various restrictions noted in Section 7.4, the compiler safely determines these sizes in most cases, but the stream networks shown in Figure 7 require more discussion. Two things may happen in these networks:

- If the *put_stream* of Loop 2 is unconditional (i.e., its enable is 1) the network computes correctly, but throughput is degraded.
- If the *put_stream* is given a computed enable, then the network may deadlock.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	50 of 241

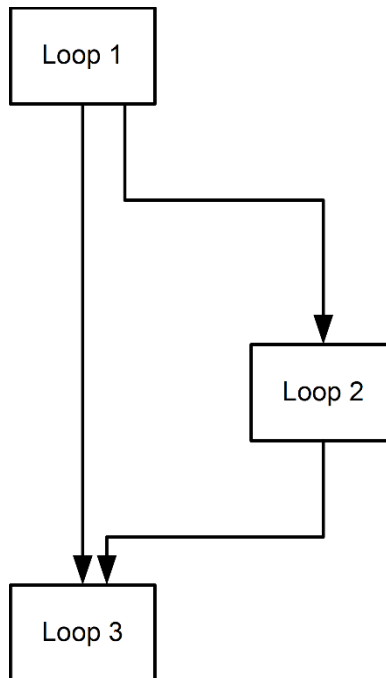


Figure 7 Stream Buffer Sizes

In the first case, using Figure 7 and assuming the enable is always 1, the *put_stream* of Loop 2 emits a value in every iteration. The compiler determines sufficient buffer sizes for correct results, but in this case they may not be sufficient for full data throughput.

- As data values flow out of Loop 1, Loop 2 consumes them.
- At this point, although receiving values from Loop 1, Loop 3 cannot yet start since it has no values available from Loop 2.
- Eventually the buffer from Loop 1 to Loop 3 fills and Loop 1 stalls.
- Meanwhile, Loop 2 is processing values and based on the pipeline depth of Loop 2, it eventually begins to produce data values.
- This allows Loop 3 to process. As it consumes data values from its stream buffers, Loop 1 is able to resume, but it is possible (depending on the pipeline depth of Loop 1) that its output buffers are empty before new values are emitted from Loop 1.
- Loop 2 and Loop 3 may have to wait before they have new data to process.

Adding stream buffer capacity to the shorter path alleviates this performance problem by allowing the path to absorb more values so that the stream producer does not have to stall. See Section 7.12 for stream buffer functions that may be used for this purpose.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	51 of 241

In the second case, using Figure 7 and assuming a computed enable input, the *put_stream* of Loop 2 may produce fewer values than it consumes. As a basic example of a computed enable, Loop 2 may emit a value only in an even-numbered iteration.

- There are twice as many values coming into Loop 3 from Loop 1 as there are from Loop 2.
- The buffer between Loop 1 and Loop 3 fills, stalling Loop 1 and depriving Loop 2 of values to read.
- Deadlock occurs.

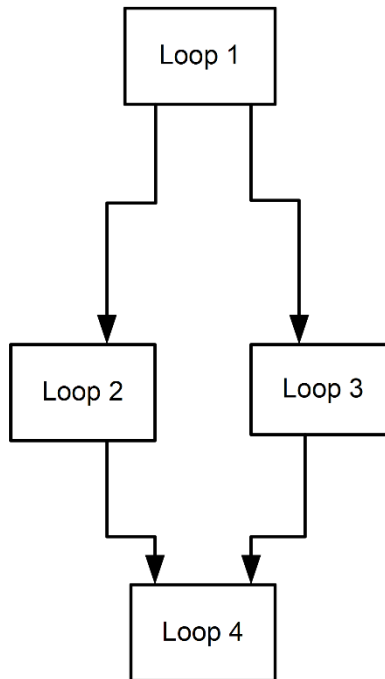


Figure 8 Stream Loops as Filters

In a slightly more serious example, Figure 8 shows a case where two loops serve as filters.

- Loop 2 has a computed enable and passes through the even values from its input stream, while Loop 3 has a computed enable and passes through the odd values.
- If Loop 1 produces sequentially incrementing data values, then the stream network works correctly regardless of the length of the streams produced by Loop 1.
- If Loop 1 produces streams in which the first 100,000 values are even, then Loop 3 produces no values, preventing Loop 4 from iterating.
- The buffer between Loop 1 and Loop 4 fills up, stalling Loop 1 and causing deadlock.

In this case, deadlock is data dependent and the compiler cannot determine an adequate buffer size.

A conditional put-stream in Loop 2 and/or Loop 3 produces a compiler warning because the compiler cannot guarantee the loops are deadlock-free.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	52 of 241

7.7 Stream Termination

In addition to the *data*, *valid*, and *stall* signals shown earlier, a stream may have a *terminate* signal. This signal originates by a call to *stream_<n>_term* where *<n>* is the number of bits in the stream.

```
Stream_32 S;
...
for (i=0; i<N; i++)
    put_stream_32 (&S, i+42, 1);
stream_32_term (&S);
```

After the last value is sent by the loop, the *stream_<n>_term* function sends a termination signal that follows the last data element. If a stream is stalled, the termination signal is guaranteed to arrive after the data. Once a stream is terminated, it is shut down for the remainder of the time that the parallel region continues to execute. Leaving the parallel region and reentering it resets all streams so that they may be used again.

The stream termination function is:

```
void stream_<n>_term (Stream_<n> *S);
```

Where *n* is the bitwidth of 8, 16, 32, 64, 128, 192, 256, 320, 512, or 1024.

Stream termination is tested in one of two ways.

1. A pipelined loop reading one or more streams may use the *all_streams_active* function to test whether all of the streams consumed by that loop are still receiving data values and have not terminated.

```
Stream_32 S0, S1;
...
while (all_streams_active()) {
    get_stream_32 (&S0, &v0);
    get_stream_32 (&S1, &v1);
    ..
}
```

The loop continues to iterate until at least one of the two input streams terminate. It is useful to understand exactly how this function works. A stream may have bubbles in its flow of data (i.e., periods of time where no values are showing up). The *all_streams_active* function waits until either a new data value arrives in the stream buffer or a termination signal is received for the stream.

Because of the way a pipelined loop is constructed, it is not possible to test the consumer streams individually. The *all_streams_active* function implicitly tests all of the consumer streams. The *all_streams_active* function is designed to work only in pipelined loops. If used in a non-pipelined loop, the loop never terminates.

Note: The call to the *all_streams_active* function may occur only in a loop termination test at the top of a loop. Other placements of this function results in undefined behavior.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	53 of 241

2. The function `is_stream_<n>_active` also tests for stream termination. This function tests an individual consumer stream. It cannot be used in the loop termination test or within a pipelined loop. Outside of pipelined loops, the function may be used freely to test individual streams for termination. The prototype for this function is:

```
int is_stream_<n>_active    (Stream_<n> *)
```

Where n is the bitwidth of 8, 16, 32, 64, 128, 192, 256, 320, 512, or 1024.

7.8 Stream Width Conversion

Sometimes it is necessary to convert from one stream width to another. Stream width converters take a stream of one width as input and produce a stream of another width as an output stream. When converting from one stream width to another stream width, there are two possibilities for ordering of the smaller stream values within the larger stream values. Consider the 64-bit hexadecimal value `0x0A0B0C0D0E0F0102`. When this value is in a 64-bit stream and the stream value is converted to an 8-bit stream, the 8-bit stream values could be returned as `0xA`, `0xB`, `0xC`, `0xD`, `0xE`, `0xE`, `0xF`, `0x1`, `0x2` or `0x2`, `0x1`, `0xF`, `0xE`, `0xD`, `0xC`, `0xB`, `0xA`. The first option is called a big-endian conversion while the second is a little-endian conversion. Both big and little endian stream width converters are provided, as are terminating converters and count based converters.

The prototypes for terminating stream conversion functions are of the form:

```
void stream_width_<n>to<m>_term (Stream_<n> *s1, Stream_<m> *s2)
```

```
void stream_width_<n>to<m>_le_term (Stream_<n> *s1, Stream_<m> *s2)
```

- *le* designates a little-endian conversion.
- n is not equal to m .
- n and $m = 8, 16, 32, 64, 128, 192, 256, 320, 512, \text{ or } 1024$.
- *s1* is the source stream.
- *s2* is the output stream.

When stream *s1* receives a termination signal, any remaining bytes in its stream buffer are converted and a termination signal is sent at the end of the data values output to stream *s2*.

The prototypes for count-based stream conversion functions are:

```
void stream_width_<n>to<m> (Stream_<N>*s1, Stream_<M> *s2, int64_t nbytes)
```

```
void stream_width_<n>to<m>_le (Stream_<N> *s1, Stream_<M> *s2, int64_t nbytes)
```

- *le* designates a little-endian conversion
- n is not equal to m .
- n and $m = 8, 16, 32, 64, 128, 192, 256, 320, 512, \text{ or } 1024$.
- *s1* is the source stream.

The variable *nbytes* is the number of bytes in both the source and output stream.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	54 of 241

9. DATA MOVEMENT

9.1 Buffered Data from the CPU

Data movement functions are added to the MAP function to move data between the SCM and the MAP processor or to move data between the VLM and the MAP processor. There are two kinds of data movement. The first uses OBM as the target destination or the source of the data. The second uses streaming so data flows in and/or out as the computation is occurring. See Section 7 for information on streaming.

The function that handles DMA operations of data between OBM banks and the CPU's memory is *buffered_dma_cpu*. This function initiates a DMA operation. The function returns after the data movement is complete:

```
void buffered_dma_cpu (int dir, int dma_path, void *obmadr, int64_t obmstride, void *cmadr,
                     int64_t cmstride, int nbytes);
```

- *dir* is CPU to On-Board Memory (CM2OBM) or On-Board Memory to CPU (OBM2CM), specifying whether the data transfer is writing to OBM from the CPU or writing to the CPU from OBM.
- *dma_path* is the DMA engine select, which is ignored for the Saturn 1 server cartridge.
- *obmadr* is the start address in OBM.
- *obmstride* is the stride argument, discussed in more detail in the following paragraph.
- *cmadr* is the CPU address.
- *cmstride* is the stride for the CPU address.
- *nbytes* is the transfer length, in bytes.

The *obmstride* argument identifies the OBM bank used for the data movement. The *obmstride* argument is generated by a call to *MAP_OBM_stripe*, which takes a stride integer and a single OBM bank string as its arguments. The function is fully evaluated at compile time. For example, the following returns a 64-bit descriptor that specifies OBM bank A with a stride of 1:

```
MAP_OBM_stripe (1, "a")
```

The prototype is:

```
int64_t MAP_OBM_stripe (int64_t stride, char *stripes);
```

There are specific constraints for this function:

- A *stripe* string may only reference a single OBM bank.
- The stride must be the constant value 1.

Example 1:

- This is the simplest and most common scenario. The DMA operation brings *num_nodes* 64-bit integers from the CPU's memory at *ND* into the MAP as the array nodes. Because the bank string in the *MAP_OBM_stripe* function has only A, all data is transferred into OBM bank A:

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	73 of 241

```
void subr (int64_t ND[], int num_nodes, int mapnum) {
    OBM_BANK_A (nodes, int64_t, MAX_OBM_SIZE)
        buffered_dma_cpu (CM2OBM, PATH_0, nodes, MAP_OBM_stripe(1, "a"),
            ND, 1, num_nodes*sizeof(int64_t));
}
```

Example 2:

- In this example, two DMA operations download separate arrays into banks A and C:

```
void subr (int64_t WX[], int64_t VX, int size, int mapnum) {
    OBM_BANK_A (ws, int64_t, MAX_OBM_SIZE)
    OBM_BANK_C (vs, int64_t, MAX_OBM_SIZE)

    buffered_dma_cpu (CM2OBM, PATH_0, ws, MAP_OBM_stripe(1, "a"), WX, 1,
        size*sizeof(int64_t));
    buffered_dma_cpu (CM2OBM, PATH_0, vs, MAP_OBM_stripe(1, "c"), VX, 1,
        size*sizeof(int64_t));
}
```

Note: Declaring the OBM banks with *MAX_OBM_SIZE* is useful in the case where the size of the incoming arrays is brought in as a parameter of the MAP function.

There may be no more than 12 lexical DMA functions in a MAP function. There is no restriction on the number of times a given function may be executed if it occurs inside a loop.

9.2 Streamed Data for the CPU

Both inbound and outbound DMA operations may be streamed. DMA streams have full flow control characteristics.

An inbound streaming DMA call is a stream producer and writes data into the stream. If the logic in the UL cannot handle the DMA's full throughput, then an inbound DMA may stall.

An outbound streaming DMA call is a stream consumer. It reads data from a stream and sends the elements out as it reads them. The DMA engine in the CC may stall an outbound DMA if the DMA engine cannot handle the throughput of data from the UL.

There is one function used for streaming 64-bit values through a DMA with the CPU:

```
void streamed_dma_cpu_64 (Stream_64 *S, int dir, void *cmaddr, int64_t nbytes);
```

- *S* is the stream.
- *dir* is the direction (*STREAM_TO_PORT* or *PORT_TO_STREAM*).
- *cmaddr* is the starting address in CPU memory.
- *nbytes* is the size of the transfer in bytes.

Streaming DMA functions move data directly into the UL as it arrives at the MAP processor. The following is an example of an inbound streaming DMA call:

```
Stream_64 S0;
```

```
#pragma src parallel sections
{
    #pragma src section
    {
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	74 of 241

```
streamed_dma_cpu_64 (&S0, PORT_TO_STREAM, In, sz*sizeof(int64_t));
}

#pragma src section
{
for (j=0; j<sz; j++) {
    get_stream_64 (&S0, &v0);
    CL[j] = v0+1;
}
}
}
```

The *streamed_dma_cpu_64* function initiates the DMA operation and fetches the data values as they arrive from the CPU, putting them into stream *S0*. The consumer loop index *j* iterates, as data values are available. If a consumer loop runs slowly due to loop dependency issues, the consumer stream and the DMA operation stall so no data values are lost.

The following is an example of an outbound streaming DMA call.

```
Stream_64 S1;
...
#pragma src parallel sections
{
    #pragma src section
    {
        streamed_dma_cpu_64 (&S1, STREAM_TO_PORT, Out, sz*sizeof(int64_t));
    }

    #pragma src section
    {
        for (j=0; j<sz; j++) {
            put_stream_64 (&S1, BL[j] + 42, 1);
        }
    }
}
```

The *streamed_dma_cpu_64* function initiates the outbound DMA operation and feeds it with data values from the stream *S1*. The producer loop automatically stalls if the DMA operation must pause during the data transfer.

9.3 Streamed Data for the VLM

There is one function used for streaming four 64-bit values through a DMA to one of the available VLM banks:

```
void streamed_dma_vlm_256 (Stream_256 *S, int dir, int vlm_bank, void *vlm_addr, int64_t
                           nbytes);
```

- *S* is the stream.
- *dir* is the direction (*STREAM_TO_PORT* or *PORT_TO_STREAM*)
- *vlm_bank* is the designated VLM bank (*VLM_BANK_0* or *VLM_BANK_1*).
- *vlm_addr* is the starting byte-address in the VLM memory.
- *nbytes* is the size of the transfer in bytes.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	75 of 241

Streaming DMA functions move data directly into the logic user chip as it arrives from the VLM bank. The following is an example of an inbound streaming DMA function:

```
Stream_256 S_read;
#pragma src parallel sections {
    #pragma src section
    {
        int i;
        for (i=0; i<m_words; i+=4)
            get_stream_256 (&S_read, &BL[i], &BL[i+1], &BL[i+2], &BL[i+3]);
    }

    #pragma src section
    {
        streamed_dma_vlm_256 (&S_read, PORT_TO_STREAM, 1, addr,
                               m_words*sizeof(int64_t));
    }
}
```

The *streamed_dma_vlm_256* function initiates a read of the values in the VLM memory starting at *addr* and transfers them to the *S_read* stream. Four 64-bit data values are read in every clock cycle, so a 256-bit stream captures all the values for each iteration.

```
Stream_256 S_write;
#pragma src parallel sections
{
    #pragma src section
    {
        streamed_dma_vlm_256 (&S_write, STREAM_TO_PORT, 1, addr,
                               m_words*sizeof(int64_t));
    }

    #pragma src section
    {
        int i;

        for (i=0; i<m_words; i+=4)
            put_stream_256 (&S_write, AL[i], AL[i+1], AL[i+2], AL[i+3], 1);
    }
}
```

The *streamed_dma_vlm_256* call initiates a write of the values in the *S_write* stream to VLM memory starting at *addr*.

9.4 Vector Stream Interface to VLM

In addition to the streaming DMA VLM interface, there is a vector streaming DMA VLM interface. Each vector represents a read or write of a contiguous block of data to the specified VLM bank. The parameters of a read or write request (address, length, and direction) are in its vector's header. This interface allows read and write requests to queue something that is not possible with the non-vector streaming interface. Two Carte C functions provide the vector stream interface. In the first function, there are three streams: write requests with data, read requests, and returning read data. Since there is no way of specifying the exact interleaving execution order of the separate write and read requests with the first function, the second function has been created to combine the read and write requests into one stream for use in cases where the order of reads and writes must be precisely deterministic.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	76 of 241

The following function handles VLM transactions using separate streams for read and write requests:

```
void vec_stream_256_vlm_write_read_term (Vec_Stream_256 *WR, Vec_Stream_256 *RD,  
                                         Vec_Stream_256 *D, int vlm_bank);
```

- *WR* is the write requests with data (a stream input).
- *RD* is the read request (a stream input).
- *D* is the read data (a stream output).
- *vlm_bank* is the designated VLM bank (*VLM_BANK_0* or *VLM_BANK_1*).

Each write request is a vector, whose header contains the write parameters and whose data are the values to be written. Each read request consists of an empty vector (a header and a tail, with no data values) whose header contains the read parameters. The read data values are returned through the third stream, with a vector of data corresponding to each read request.

The following function handles VLM transactions using a combined read/write request stream:

```
void vec_stream_256_vlm_term (Vec_Stream_256 *RQ, Vec_Stream_256 *D, int vlm_bank);
```

- *RQ* is the read requests and write requests with data (a stream input).
- *D* is the read data (a stream output).
- *vlm_bank* is the designated VLM bank (*VLM_BANK_0* or *VLM_BANK_1*).

The function has two streams. One stream handles requests and writes data and one stream returns read data. The general principles are the same as for the previous function. Both of these functions terminate when their request input streams terminate.

As was mentioned earlier, the request parameters are in the headers of the requests. A header can be created explicitly, using the *put_vec_stream_256_header* function for putting headers on vector streams, or a pair of special functions can be used. For the explicitly created manual header, the four 64-bit fields that make up the 256-bit header hold the address, length, user tag, and direction (i.e., read or write). The tag has no meaning for writes, but will show up in the header of the return read data, which helps in identifying which read request the data vector is associated with. This function is already documented in the general vector stream section of this document, but as a reminder, here is its prototype used to make a VLM vector header:

```
void put_vec_stream_256_header (Vec_Stream_256 *RQ, int64_t vlm_addr, int64_t length, int64_t  
                               tag, int64_t vlm_cmd);
```

- *RQ* is the request stream being written to.
- *vlm_addr* is the starting byte address.
- *length* is the size of the transfer in bytes.
- *tag* is arbitrary vector stream header data.
- *vlm_cmd* is *VLM_WRITE_CMD* or *VLM_READ_CMD*.

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	77 of 241

There are two special functions that do the same thing; they are specifically designed for VLM read and write requests.

```
void put_vlm_write_header (Vec_Stream_256 *RQ, int64_t vlm_addr, int64_t length);
```

```
void put_vlm_read_command (Vec_Stream_256 *RQ, int64_t vlm_addr, int64_t length, int64_t tag);
```

- *RQ* is the request stream being written to.
- *vlm_addr* is the starting byte address.
- *length* is the size of the transfer in bytes.
- *tag* is arbitrary vector stream header data.

The following Carte C function demonstrates how to use the vector stream interface to VLM.

```
// Write an addressing pattern to VLM using vector streams
// Copy data from VLM to OBM and then back to CPU memory

#include <libmap.h>

void
subr (int64_t Res[], int sz, int iter, int mapnum)
{
    OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)

    Vec_Stream_256    WR, RD, RQ, D;
    Stream_64         S0;
    int64_t            num_vals=0;

    #pragma src parallel sections
    {
        #pragma src section
        {
            vec_stream_256_vlm_term(&RQ, &D, VLM_BANK_0);
        }

        #pragma src section
        {
            vec_stream_merge_nd_2_256_term(&RD, &WR, &RQ);
        }

        //
        // Write addressing pattern to VLM and read it back
        //
        #pragma src section
        {
            int        i, j;
            int64_t     adr, len, tag;

            adr = 0x00ULL;
            len = sz*4*sizeof(int64_t);           // number of bytes
            for (i=0; i<iter; i++) {
                tag = 0xF0F0F0F0;
                printf("put_vec_stream_256_header write0 %ld: %ld\n", adr, len);
            }
        }
    }
}
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	78 of 241

```

        put_vlm_write_header (&WR, adr, len);
// Alternate method
// put_vec_stream_256_header (&WR, adr, len, tag, VLM_WRITE_CMD);
for (j=0; j<sz; j++) {
    put_vec_stream_256 (&WR,
        (int64_t) (4*sz*i + 4*j + 0),
        (int64_t) (4*sz*i + 4*j + 1),
        (int64_t) (4*sz*i + 4*j + 2),
        (int64_t) (4*sz*i + 4*j + 3), 1);
}
    put_vec_stream_256_tail (&WR, 0, 0, 0, 0);

    put_vlm_read_command (&RD, adr, len, ~tag);
    adr += len;
}
vec_stream_256_term (&WR);
vec_stream_256_term (&RD);
}
//
// VLM Read Data Stream
//
#pragma src section
{
    int i, j, cnt;
    int64_t head[4], tail[4];
    int64_t val_0, val_1, val_2, val_3;

    cnt = 0;
    // Respond to TERM
    while (is_vec_stream_256_active (&D)) {
        get_vec_stream_256_header (&D, &head[0], &head[1], &head[2], &head[3]);
        j = head[0]/8;

        // Respond to TAIL
        while (all_vec_streams_active()) {
            get_vec_stream_256 (&D, &val_0, &val_1, &val_2, &val_3);
            AL[j+0] = val_0;
            AL[j+1] = val_1;
            AL[j+2] = val_2;
            AL[j+3] = val_3;
            j+=4;
            cnt+=4;
        }
        get_vec_stream_256_tail (&D, &tail[0], &tail[1], &tail[2], &tail[3]);
    }

    num_vals = cnt;
}
}
#pragma src parallel sections
{
    #pragma src section
    {
        streamed_dma_cpu_64 (&S0, STREAM_TO_PORT, Res, num_vals*sizeof(int64_t));
    }
}

```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	79 of 241

```
#pragma src section
{
    int i;
    for (i=0; i<num_vals; i++) {
        put_stream_64 (&S0, AL[i], 1);
    };
}
}
```

Doc No.	Rev.	Title	Date	Page
69266	3	Saturn 1 Carte™ C Programming Environment Guide	2 Feb 16	80 of 241