Dataflow Programming for Networking

Version 2014.2

MAXELER
Technologies
MAXIMUM PERFORMANCE COMPUTING

Dataflow Programming for Networking. Copyright © Maxeler Technologies.

Version 2014.2
January 25, 2015

Contact Information

Sales/general information: info@maxeler.com

US Office

Maxeler Technologies Inc
Pacific Business Center
2225 E. Bayshore Road
Palo Alto, CA 94303, USA.
Tel: +1 (650) 320-1614

UK Office

Maxeler Technologies Ltd
1 Down Place
London W6 9JH, UK.
Tel: +44 (0) 208 762 6196

All rights reserved. The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this document may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the written permission of the copyright holder.

Maxeler Technolgies and the Maxeler Technologies logo are registered trademarks of Maxeler Technologies, Inc. Other product or brand names may be trademarks or registered trademarks of their respective holders.

r44158

Preface

High performance network applications pose special challenges for developers, beyond the usual demands of high performance computing. Success is measured not only by the speed of calculations but by the ability to deliver results reliably with minimal latency and jitter with consistently high levels of throughput. As market competition has continued to necessitate pushing the envelope, the dataflow computing model supported by Maxeler platforms has become the method of choice for many network applications.

The Maxeler tool chain alleviates some of the inherent complexity of dataflow application development by decoupling the code for the computational building blocks (**Kernels**) from that which specifies the interconnections among them (**Managers**). In addition, support for ubiquitous protocols such as TCP and UDP allow developers to focus on higher-level functionality. These aspects of the application are expressed in MaxJ, whereas other functions outside the critical path such as initialization and logging are implemented in C or C++ code running on the CPU.

This document serves as a companion volume to **Multiscale Dataflow Programming**. Some overlap in subject matter allows this document to be self-contained for the purpose of working with network applications. For other aspects of high performance scientific or numerical computing please consult both volumes.

Document Conventions

When important concepts are introduced for the first time, they will appear in **bold**.
Italics are used for emphasis.
Directories and commands are displayed in `typewriter` font.
Variable and function names are displayed in `typewriter` font.
MaxJ methods and classes are shown using the following format:

*DFEVar **mem**.romMapped(**String** name, DFEVar addr, DFEType type, **double**... data)*

C function prototypes are similar:

*max_engine_t∗ max_load(max_maxfile_t∗ maxfile, **const char**∗ engine_id_pattern);*

Actual MaxJ usage is shown without italics:

io.output("output", myRom, dfeUInt(32));

C usage is similarly without italics:

engine = max_load(maxfile, engine_id_pattern);

Sections of code taken from the source of the examples appear with a border and line numbers:

```
package chap01_gettingstarted.ex1_passthrough;
import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

public class PassThroughKernel extends Kernel {
    protected PassThroughKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar x = io.input("x", dfeUInt(32));
        // Output
        io.output("y", x, dfeUInt(32));
    }
}
```

1 Network Programming Overview

Traditional system architectures involving Network Interface Cards (NICs) added into CPU systems put hard lower bounds on latency, and introduce a large amount of variability (jitter) into a network application's performance.

Putting the entire critical path of an application into a **Dataflow Engine** (DFE) allows programmers to take advantage of ultra-fine-grained and pipelined parallelism, enabling different data to undergo different operations at the same time, much like an assembly line in a factory.

As opposed to a traditional time-slicing CPU architecture, dedicated channels directly connecting the processing to the network guarantee that data moves at the right time and rate. Data is not delayed by unnecessary transfers through main memory for processing by a software protocol stack, nor by waiting for service from operating system interrupt handlers.

Despite the big gains from the **Dataflow** approach, conventional programming languages and technologies are also employed on Maxeler Dataflow platforms, making them a productive alternative for critical environments using existing skill sets. The remainder of this chapter summarizes the key concepts and performance benefits available when developing network applications.

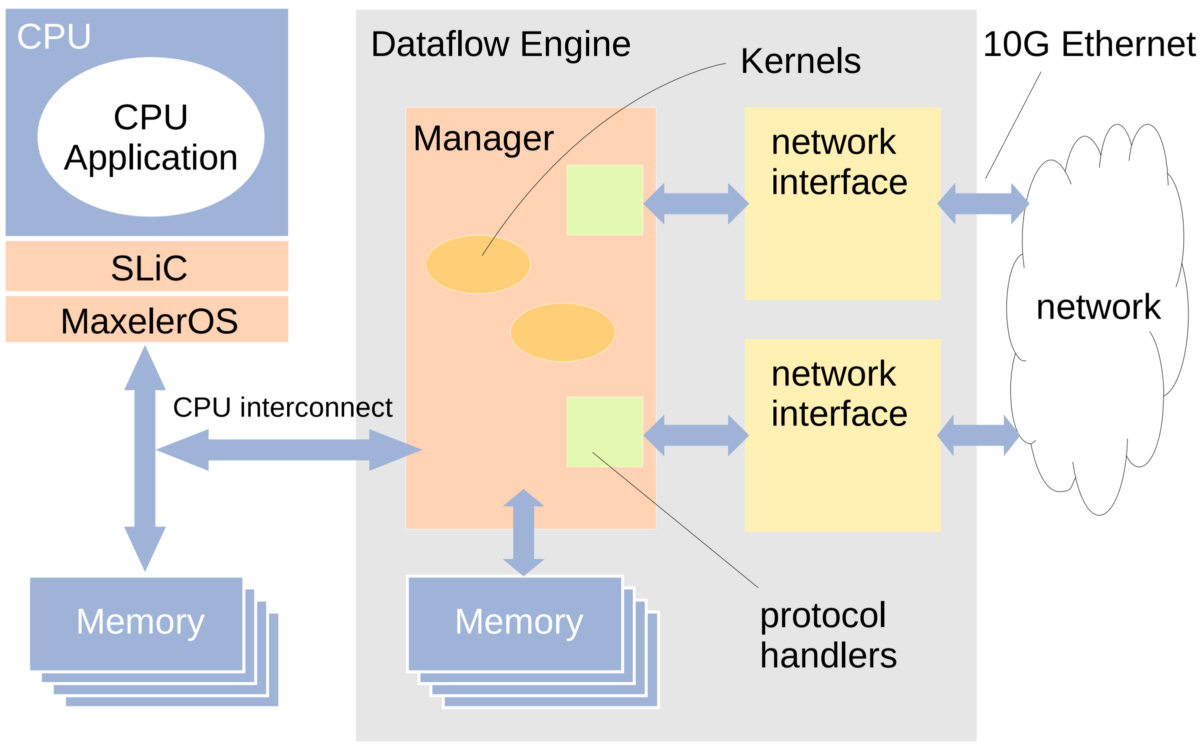


Figure 1: Maxeler network infrastructure

1.1 Maxeler Networking Technology

Figure 1 shows the architecture of a network application when implemented on a DFE, including:

- A conventional CPU is still present, but now has a DFE, which is directly connected to the network.
- Data can follow a path from the network, through the DFE, and back to the network without involving the CPU at all.
- The **Manager** within the DFE is customized with a combination of user-defined computational **Kernels** and automatically generated protocol handling blocks that interface directly with fast networking hardware.

The ideal case is where the entirety of the application's critical-path can be contained in the DFE, using the CPU for "housekeeping" tasks. Although a moderate volume of network data can be transferred between the DFE and the CPU over the CPU interconnect channel, this transfer takes time and risks becoming a bottleneck to the overall application, especially once the vagaries of CPU scheduling is taken into account.

1.1.1 Hardware

DFE hardware is designed to allow high speed computation and networking.

- **Memory** The DFE includes up to 48 GB of on-board memory accessible independently of the CPU main memory by a high-bandwidth bus.
- **Network interfaces** There are two high-performance network interfaces per DFE, each having an industry standard SFP+ port capable of 10 Gb/s transfer rates.

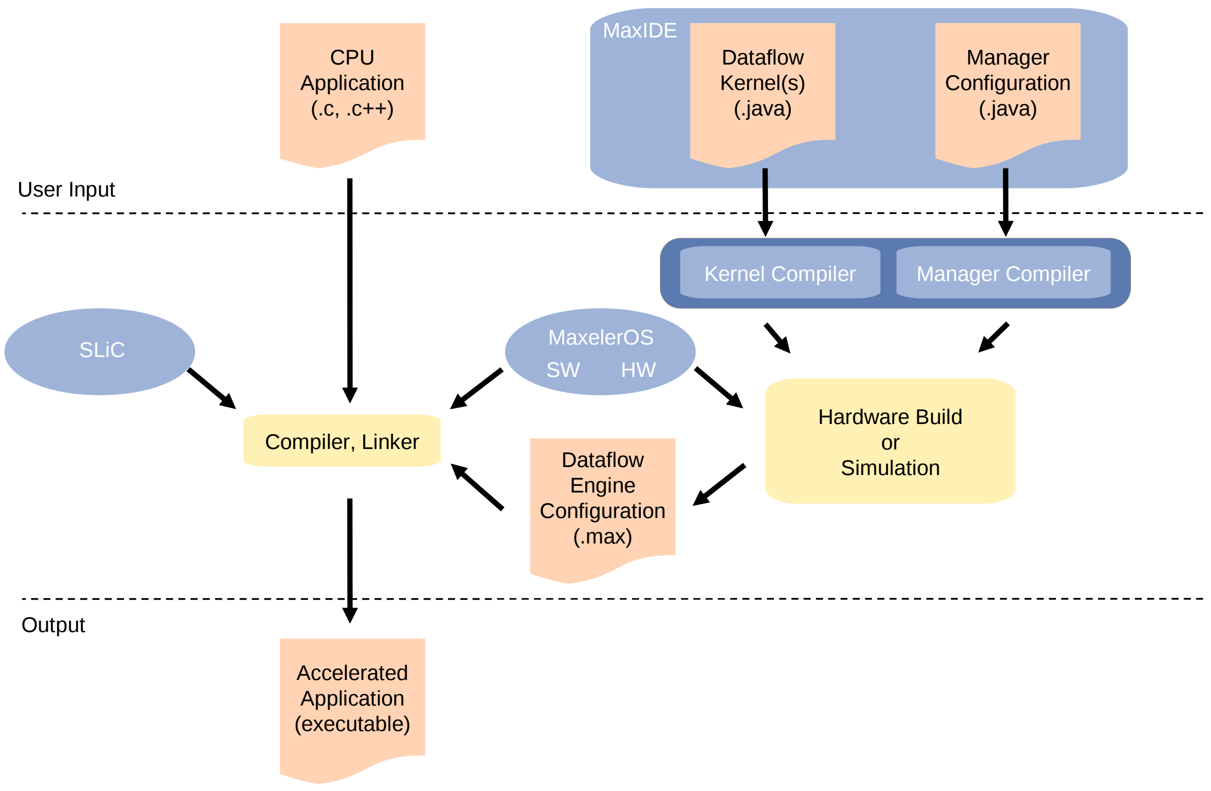

CPU Application (.c, .c++)
MaxIDE
Dataflow Kernel(s) (.java)
Manager Configuration (.java)
User Input
Kernel Compiler
Manager Compiler
SLiC
MaxelerOS
SW
HW
Compiler, Linker
Hardware Build or Simulation
Dataflow Engine Configuration (.max)
Output
Accelerated Application (executable)

Figure 2: Maxeler software component interactions

1.1.2 Software

Maxeler provides a set of tools including MaxCompiler, presenting an agile environment for network application developers. *Figure 2* shows some of the development tools provided by Maxeler and how they interact to build an accelerated application.

- **SLiC** is a C-language library usable by applications written in C, C++, and other languages, that enables them to communicate with a DFE, and to control its network interfaces as necessary to manage connections and addresses.
- **MaxelerOS** is used internally by MaxCompiler to provide the link between the CPU and the dataflow engines of an accelerated application. MaxelerOS runs on every node with a Maxeler dataflow system, providing drivers for the dataflow engines and IP cores facilitating the communication between the CPU and dataflow engines.
- **MaxCompiler** compiles a design for the Kernels and Managers that determine the function of a DFE.
- **MaxIDE** is an integrated development environment based on the open source Eclipse platform, with additional features supporting MaxCompiler.
- **MaxCompilerSim** (Detailed in *Figure 3*) facilitates debugging during development by creating a simulated network environment.

Specific features of the compiler and the simulator relevant to network application development are elaborated briefly below.

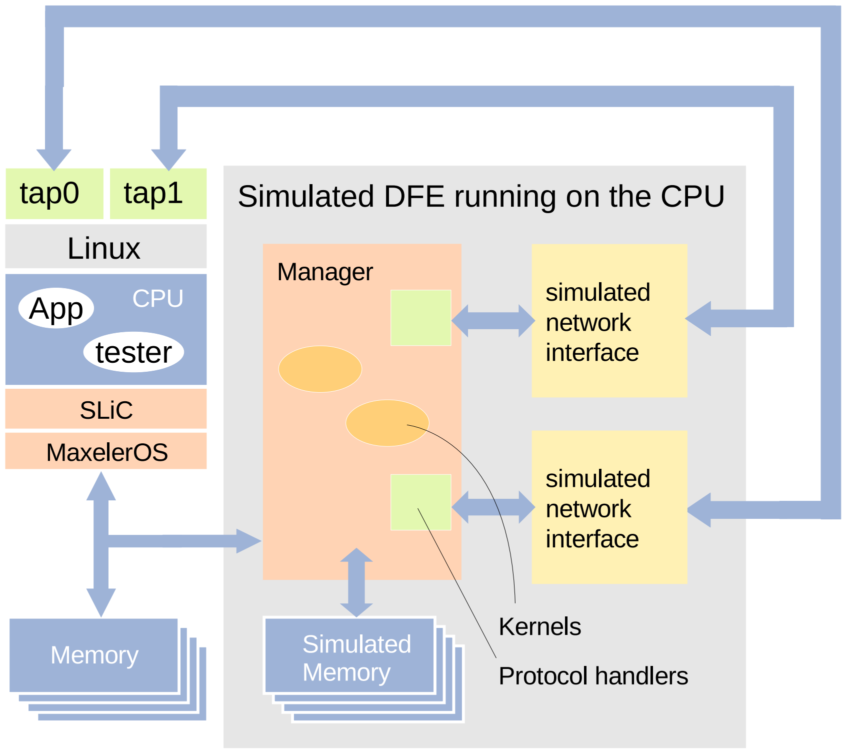


Figure 3: the simulated network created by MaxCompilerSim, with point-to-point connections from the Linux TAP devices to the network interfaces on the simulated DFE (cf. *Figure 1*)

Framed Kernels In network applications, **Framed Kernels** are a enhanced Kernel used to process data frames arriving at unpredictable times with various sizes and formats. In the MaxJ source code for a Framed Kernel, a frame appears as a structure with an assortment of fields that are easily accessible.

Framed Kernels insulate the developer from the details of unpacking and marshalling data in standard protocol formats without compromising on performance. Highly optimized protocol handling is handled by the manager to allow communication with other hosts.

Network simulation When simulating a DFE, MaxCompilerSim works in conjunction with the standard Linux TAP/TUN facility to create a simulated network. The simulated network has a point-to-point connection between a Linux TAP device and a network interface on the simulated DFE, similar to a crossover cable between two physical Ethernet ports, as shown in *Figure 3*. When the application and the simulated DFE are running, any process designed to test the application can run on the same CPU and interact with it as a remote peer would, with traffic automatically routed via the TAP devices.

1.2 Advantages of combining the dataflow model of computation with networking

Differences between the CPU model and the dataflow model of computation have significant implications for performance and efficiency in network applications. These are summarized in this section from a developer's point of view in reference to Maxeler platforms.

1.2.1 CPU Model

When implementing a software application for a CPU, the developer typically writes in a high level programming language that is compiled into a list of instructions. The instructions control the operation

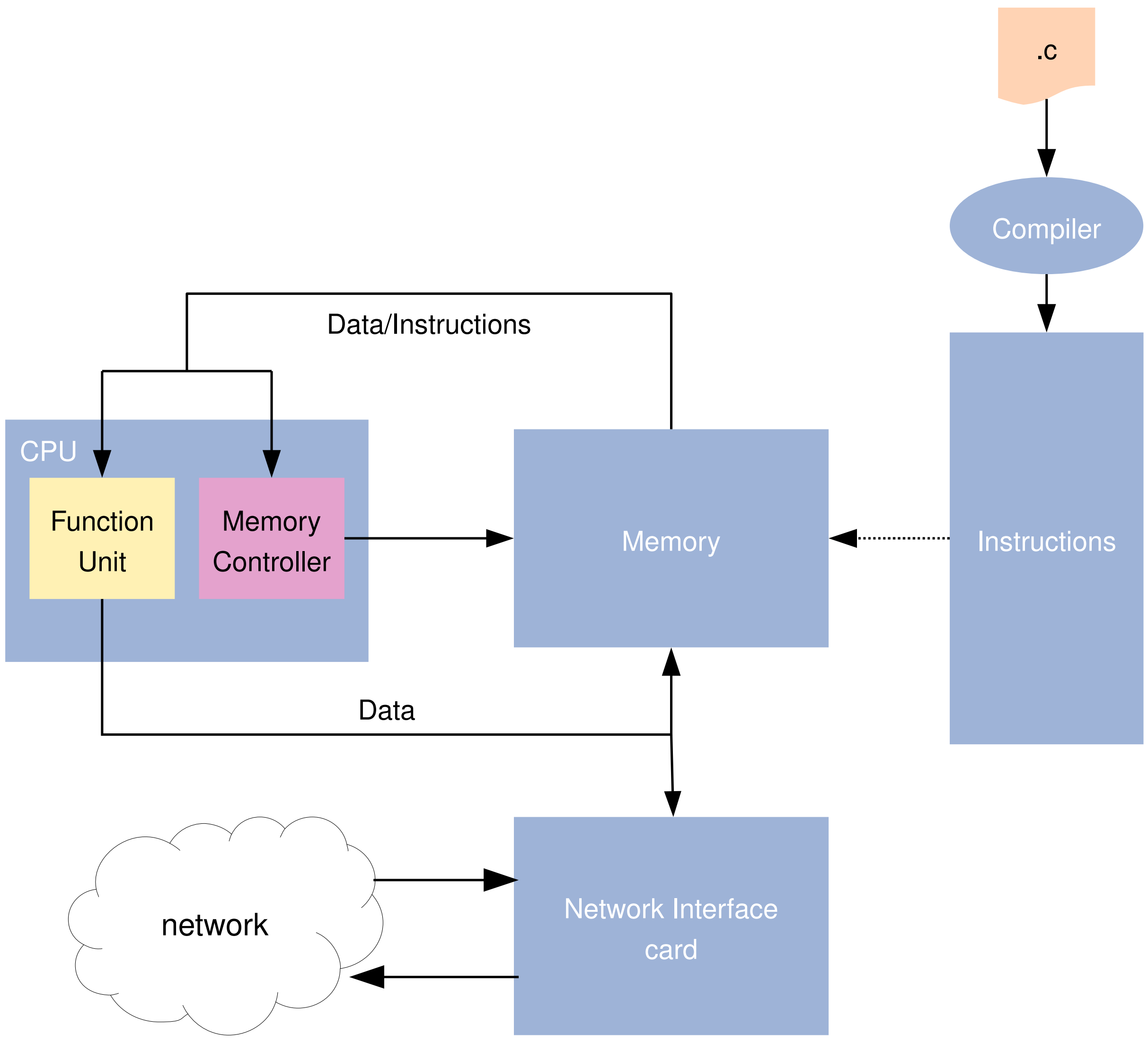


Figure 4: Reuse of functional units over time in a CPU

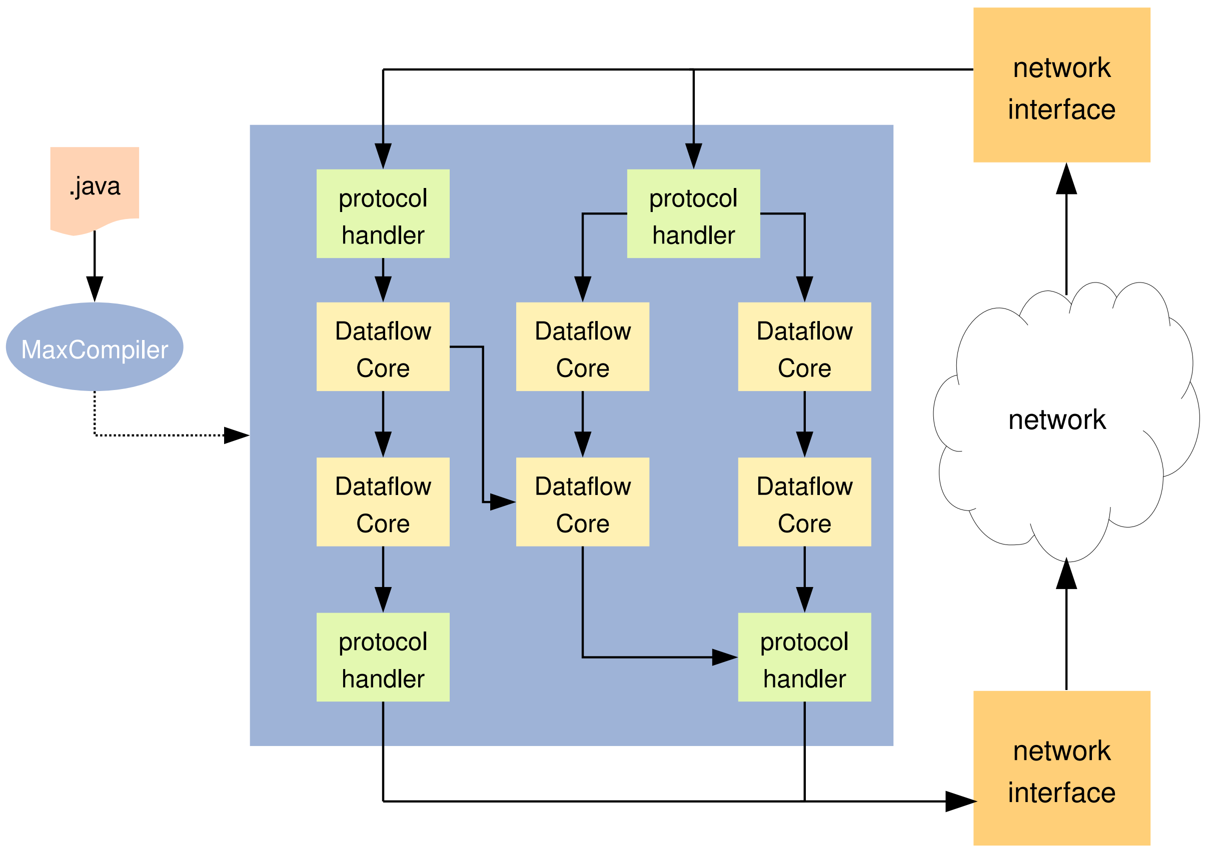


Figure 5: Graph of dedicated functions in a dataflow implementation

of functional units within the CPU. These functional units form a fixed architecture to be reused over time, and are the only available means of performing any calculations or transformations on the data. At any given time, the great majority of the data is stored idly in memory, while the CPU operates on one individual item. Transfers between the CPU and memory are required for each operation on an item of data (whether contemporaneously or amortized).

Access to the network in the CPU model is through a network interface card whose sole function is to transfer raw data between the network and memory, invariably with further intervention by the operating system required in software. This organization is depicted in *Figure 4*.

1.2.2 Dataflow Model

Maxeler platforms implement a **dataflow model of computation**. Computations are described structurally (computing in space) rather than being described by a sequence of processor instructions (computing in time).

In this model of computation, a high level language is used to generate a **graph** of operations. Each node in the graph executes a specific function on incoming data and outputs the result, which becomes the input to another node in the graph. The data flows from the network to the graph and back to the network, without needing to be written to memory. An example of this organization is shown in *Figure 5*.

The edges connecting the nodes in a dataflow graph and carrying data between them are appropriately called **streams**. Because the hardware implementation of a stream is often no more than a simple point-to-point bus, multiple streams are easily configured to carry data concurrently between functional units, with the bus widths customized to fit the data they carry. Throughput is not necessarily

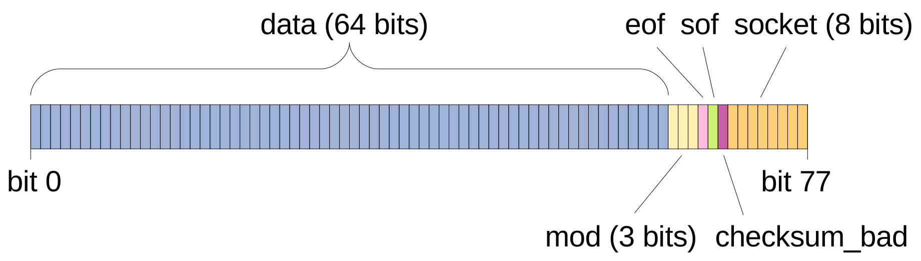


Figure 6: A stream format for incoming UDP data

constrained by the limitation to a single overburdened channel between the CPU and memory, as in the case of a CPU architecture.

1.2.3 Stream Formats for Network Data

In network applications, the specification of streams is not always trivial. Data arrives in frames is varying length, but must be transferred within the DFE over fixed-width **streams**. This is achieved by splitting the frame up into fixed-width words (according to the size of the **stream**). Each word is then accompanied with some metadata, such as address information, length, checksum validity, and Start-Of-Frame (SOF) and End-Of-Frame (EOF) markers.

MaxCompiler defines stream formats for several widely used network protocols. An example of a pre-defined stream format is shown in *Figure 6*. This format is specific to UDP traffic received from the network. Although the payload data field in the stream is of a fixed size, data of any convenient size or format is transparently marshalled on behalf of applications using the Framed Kernel interface.

1.3 MaxCompiler Design Flow

Figure 7 shows the design flow for implementing a dataflow application using MaxCompiler. This section describes the main stages in detail.

1.4 Identifying Critical Paths for Dataflow Engine Implementation

The first step in accelerating any application is to analyze the application source code to identify the parts that should be implemented in a dataflow engine. This choice is driven by the need to minimize the **latency**, or the turnaround time for a transaction between the application and a remote peer across the network.

- **Critical path operations** As noted in *subsection 1.1*, the latency is most improved by ensuring that the critical path is implemented entirely within the DFE. Consequently, accelerating a network application entails identifying the minimal sequence of steps needed to dispatch a correct response, and implementing those in terms of Kernels or other DFE components. If supplementary data or tables are needed to complete a transaction, for example, it is best to locate those in the on-board memory rather than the CPU main memory or file system.

- **Non-critical path operations** An application may be required to perform certain operations that are necessary and important, yet not an absolute prerequisite to the completion of any pending transaction. Examples include maintaining security logs or legal records. These operations belong off the DFE and on the CPU, preferably with only infrequent co-ordination between them

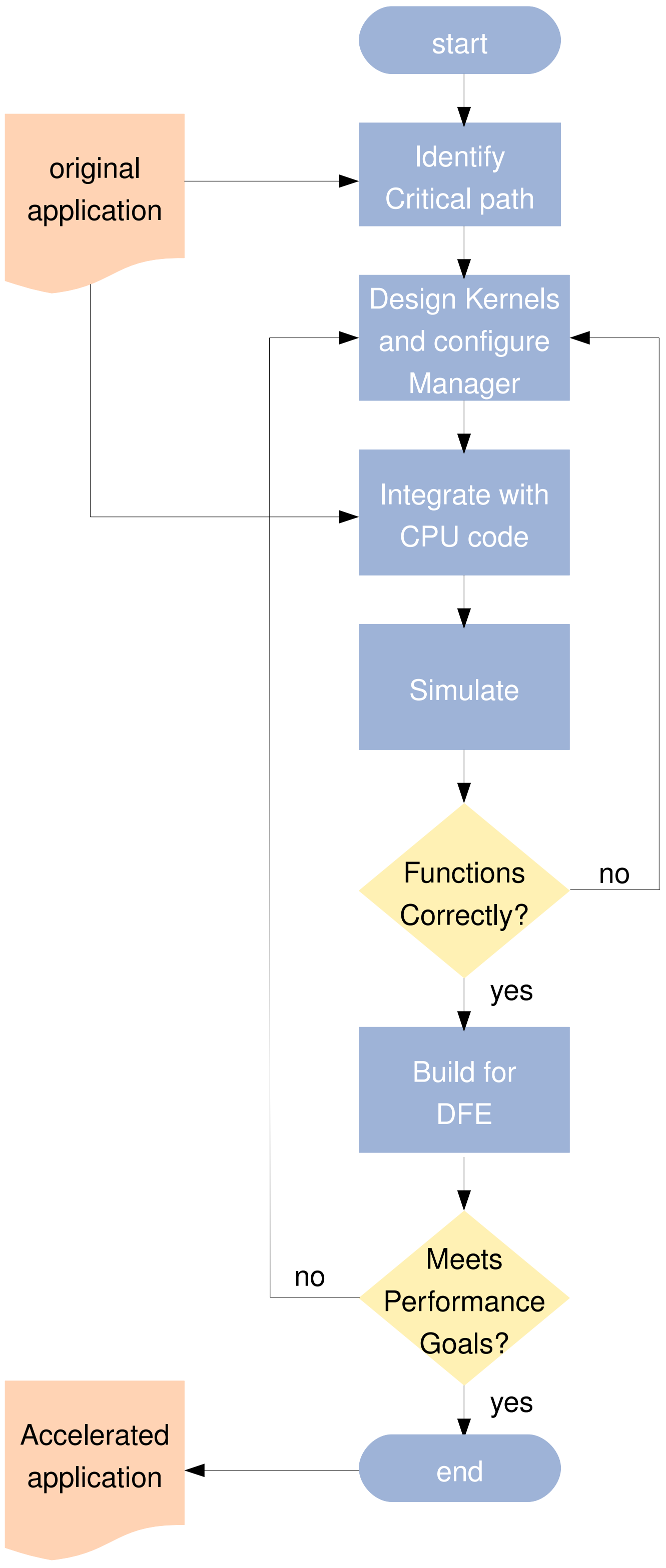


Figure 7: Flowchart of the design methodology

via the CPU interconnect. Any application also requires some amount of setup and initialization, all of which should be performed by the CPU to optimize run-time performance (for example, by building efficient storage retrieval data structures to be loaded in advance into the on-board memory if needed).

After the critical path and non-critical path operations are identified and implemented accordingly, some scope may remain for accelerating the application further by turning to more traditional areas of improvement. Dataflow engines outperform software implementations most significantly when performing the same operation repeatedly on many data items, for example, when computing an inner loop. Small areas of code that account for large shares of CPU time, whether network related or not, are suitable candidates for a DFE implementation.

1.4.1 Designing Kernels

To create Kernel designs, MaxCompiler includes the **Maxeler Kernel Compiler**. The Maxeler Kernel Compiler is a high-level programming system for creating high-performance logic and arithmetic circuits. Leveraging the power of the MaxJ language, it enables complex and parameterized Kernel designs to be mapped to application-specific circuit implementations.

The Kernel Compiler itself is implemented as a MaxJ library. To create a Kernel design, the developer writes MaxJ programs that make calls using the Kernel Compiler API. Although computations are expressed by making calls to a library, arithmetic operations are often expressed in a similar style to that of C or other standard languages, because the Kernel Compiler API uses operator overloading where applicable. (This feature is a MaxJ extension to the standard MaxJ language.)

An advantage of interfacing with the Kernel Compiler through a MaxJ library is that the full programming power of MaxJ can be deployed to customize a design. For example, by using standard MaxJ constructs such as `if`-statements and `for`-loops, developers can control the way a design is built to make optimal use of resources. Different dataflow implementations can be built from a common Kernel code base optimized for different operating scenarios. Because the Kernel design can be expressed in terms of MaxJ classes and procedures, MaxCompiler programs support extensive code reuse across different applications.

1.4.2 Configuring a Manager

Kernels are integrated into a design and connected to data channels outside the DFE by a Manager. Similarly to the Kernel Compiler, the Manager Compiler is accessed through a MaxJ API.

MaxCompiler offers a choice of Managers that can be parameterized, some for general purpose applications and others for more specific needs. Among these, the **Standard Manager** and the **Network Manager** are most relevant to network applications.

- The Standard Manager provides for a single Kernel with some implied streams connecting it to the CPU and/or the network interfaces and memories. It is suitable for rapid deployment of simple network applications.

- The Network Manager supports any number of Kernels as well as other more advanced features such as **state machines**. More general internal DFE topologies are possible than in the Standard Manager. The Network Manager is suitable for complex network application development.

When the MaxJ code for Kernels and the configuration of a Manager are combined, they form a complete MaxJ program. The execution of this program results in the generation of a dataflow engine configuration file (with a `.max` suffix) as shown in *Figure 2*.

1.4.3 Compiling

Due to the Kernel Compiler (and other elements of MaxCompiler) being accessed as a MaxJ library, there are several different stages of compilation:

1. The first stage is **MaxJ Compilation** of the Kernel and Manager source code, with normal MaxJ syntax checking, etc.. The successful result of this stage is one or more MaxJ `.class` files.
2. The next stages of compilation take place at **MaxJ run-time**, when the MaxJ code in the compiled `.class` files is executed. This process encapsulates the following further compilation steps:
   (a) **Graph construction**: In this stage, a graph is constructed in memory based on the calls made by the Kernel and Manager code using the Kernel Compiler API.
   (b) **Kernel compilation**: After all the code to describe a design has been executed, the Kernel Compiler takes the generated graph, optimizes it, and converts it either to a simulation model, or a format suitable for generating a dataflow engine.
   (c) **Back end compilation**: For a hardware build, at this point third-party vendor tools are called automatically by MaxCompiler to compile the design further into a chip configuration.

1.4.4 Integrating with the CPU Application

SLiC and the application are linked against the `.max` file generated by MaxCompiler by a standard software compiler suite such as gcc. MaxelerOS is then able to automate the configuration of dataflow engines and to provide the communications between the CPU software and the DFE implementation.

To work together with the DFE, the CPU application requires modifications such as the following from its original form.

- At a minimum, SLiC library function calls are added for configuring, starting, and stopping the DFE.
- Some of these function calls are concerned with initializing network ports and addresses, and replace any standard operating system socket API functions used in the original version.
- Critical-path business logic taken over by the DFE in the accelerated version is removed from the CPU code.
- If any data is to be transferred between the CPU and the DFE through the CPU interconnect channel, SLiC function calls for initializing and communicating by named streams are added. These modifications might replace parts of the original version where pointers or arrays are passed as parameters to functions formerly implemented in the CPU code.

1.4.5 Simulating

Network applications can be developed rapidly by being built and tested in simulation. The simulator supports a virtual network in software with point-to-point connections between the network interfaces on the simulated DFE and the system TAP devices, as shown in *Figure 3*, so that a process running on the same CPU as the application can test it by playing the role of a remote peer.

Simulation offers the advantage of visibility into the execution of a Kernel that is not available in a hardware implementation. A further advantage of simulation is a much shorter build time than for a hardware implementation.

Simulation can confirm functional correctness but not performance properties. Because the simulation of a design runs much more slowly than a real implementation, hardware should be used later in the development cycle to test the design with realistic network traffic.

1.4.6 Building for DFEs

Executing a hardware Manager generates a DFE configuration file with a `.max` suffix. This file contains both data used to configure the DFE and metadata used by software to communicate with this specific DFE configuration. The build process can take many hours for a complex design, so simulation is recommended for early verification of the design.

1.5 A Look at Framed Kernels

As noted previously, the basic units of computational activity in Maxeler accelerated applications are known as Kernels. A Kernel expresses functionality on the level of mathematical operations (e.g., addition and multiplication), control elements (e.g., counters and multiplexers), and stream IO operations. This description is mapped to interconnected dataflow cores by MaxCompiler, which a DFE implements.

While many robust techniques and methodologies for implementing mathematical algorithms with Kernels can be found in the companion to this document, **Multiscale Dataflow Programming with MaxCompiler**, this section focuses on a more immediate problem for network applications: efficient and effective network processing.

MaxCompiler provides built-in support for standard communications protocols (Ethernet, TCP, and UDP), with associated libraries describing the stream formats associated with them (*subsubsection 1.2.3*). However, these features by themselves do not address the issue of conveniently transporting user-defined data types by these standard protocols, as almost any non-trivial network application requires. Potential programming complexities include:

- packing and unpacking variable sized data for transport by fixed-width DFE links
- buffering frames over multiple cycles because a decision depends on the last field to be received
- acquiring expertise on low-level implementation details such as metadata field semantics

These issues are addressed by **Framed Kernels**.

1.5.1 Framed Kernel Basics

In a Framed Kernel, network data is considered as a stream of “frames”. Frames are typically received on a Kernel input stream, processed in a suitable way and transmitted through a Kernel output stream. Each frame consists of a fixed set of fields. Fields have familiar types, such as floating point or integer numbers and may be fixed or variable size.

The order of the fields in a frame as well as their type and name is typically application-dependent and defined by the developer as a “frame format” which can be associated with Kernel input or output streams.

A frame format may describe frames whose total size exceeds the width of the carrier stream. In this case, sufficiently small segments of the frame are streamed sequentially and MaxCompiler automatically manages the marshalling and unmarshalling of the segments.

1.5.2 Network Streams

Frame formats can be defined independently of any network protocols. Only when associating a frame format with a Kernel input or output stream the developer specifies a network protocol by selecting a pre-defined “framed link type”. MaxCompiler currently offers a choice UDP, TCP or raw Ethernet framed link types. *section 5*, *section 6* and *section 8*, respectively, describe the framed link types in greater detail.

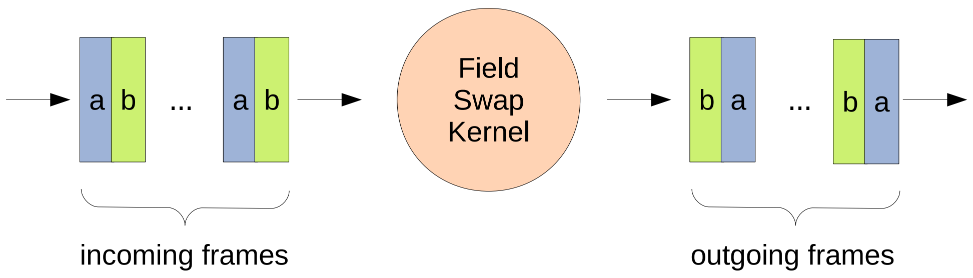


Figure 8: A stream of data in a user-defined frame format has its fields interchanged by a nearly protocol-agnostic field swapping Kernel.

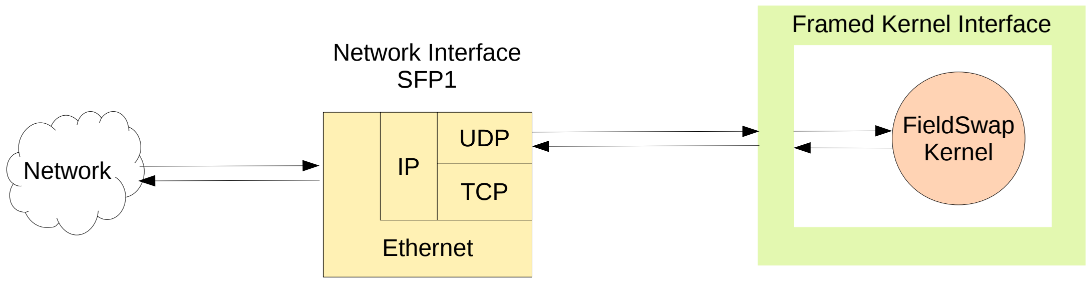


Figure 9: The field swap Manager connects the Kernel input and output directly to the UDP streams of network interface “SFP1”. DFE Link format conversions are handled by MaxCompiler automatically.

Choosing a framed link type for a stream implies that a UDP stream, for example, can be connected to a network interface in the Manager, allowing remote Kernels to exchange data with hosts using standard network infrastructure.

Depending on the application and network protocol used, some protocol-specific handling may be needed. For example, UDP packets originating from multiple possible peers contain a “socket” field to distinguish among them. The Framed Kernel interface supports methods for manipulating such protocol-specific metadata.

1.5.3 Framed Kernel Example: Field Swap

A very simple example of something that can be done by a Framed Kernel is shown in *Figure 8*. This Framed Kernel takes in frames that have two fields. For each frame received on the input, the Framed Kernel transmits a frame on the output that has the same fields, but in the opposite order. A similar technique might be used in an application such as an echo server that works by swapping the source and destination address fields in every packet.

Although the field swap Kernel operates for the most part on a user-defined frame format, the framed data is actually tunneled through UDP. *Figure 9* shows the field swap Manager which connects the Kernel’s input and output UDP streams directly to a network interface. The conversions between the user-defined frame format, the UDP framed link type and Ethernet frames is handled by MaxCompiler automatically.

The following two sections explain the implementation of the field swap Kernel and Manager in greater detail. For in-depth information Framed Kernels please refer to Chapter 3.

1.5.4 Field Swap Kernel

Swapping two fields in a Framed Kernel is very simple and the bulk of the work can be done just by these lines.

```
        frameOut["field_a"] <== frameIn["field_b"];
        frameOut["field_b"] <== frameIn["field_a"];
```

The <== is read as the **connection** operator. This code says that input field `b` is connected to output field `a` and vice versa.

To make this formulation possible, a frame format with the two fields `field_a` and `field_b` is declared as shown.

```
    private static class TestFrameFormat extends FrameFormat {
        public TestFrameFormat() {
            super(ByteOrder.LITTLE_ENDIAN);

            addField("field_a", dfeUInt(32));
            addField("field_b", dfeUInt(32));
        }
    }
```

An input stream carrying data of this format is declared like this.

```
        TestFrameFormat testFrameFormat = new TestFrameFormat();

        FrameData<TestFrameFormat> frameIn =
            io.frameInput("frameIn", testFrameFormat, new UDPOneToOneRXType());
```

Because the underlying protocol is UDP, the stream contains a "socket" field, as noted above. The socket field is not automatically set by the framed interface, so it is done explicitly. In this example, the destination socket is set to the same as the source socket, which provides for the modified frames to be routed back to their sender. Socket numbers are associated with UDP streams in the CPU code which is omitted here for brevity.

Finally, the following lines declare and connect an output stream that is able to transmit data any time the input is fully available.

```
        io.frameOutput("frameOut", frameOut);
```

The complete listing of the Kernel source code is shown in *Listing 1*. The Manager code for this example is explained in more detail in the next section.

1.5.5 Field Swap Manager

The only task left to do is to connect the field swap Kernel to a network interface, in order to allow it to communicate with remote peer. The code for this is shown in *Listing 2*. The Standard Manager is sufficient for the field swap design, because it has only a single Kernel, which is set by these lines.

```
        Kernel k = new FieldSwapKernel(m.makeKernelParameters("fieldSwapKernel"));
        m.setKernel(k);
```

The UDP streams connecting the Kernel to the network interface depicted in *Figure 9* are declared as follows.

```
        m.setIO(link("frameIn",  UDP(NetworkConnection.CH2_SFP1, UDPConnectionMode.OneToOne)),
                link("frameOut", UDP(NetworkConnection.CH2_SFP1, UDPConnectionMode.OneToOne)));
```

Listing 1: Field swap example Kernel (FieldSwapKernel.maxj)

```
package fieldswap;

import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.networking.v1.framed_kernels.ByteOrder;
import com.maxeler.networking.v1.framed_kernels.FrameData;
import com.maxeler.networking.v1.framed_kernels.FrameFormat;
import com.maxeler.networking.v1.framed_kernels.FramedKernel;
import com.maxeler.networking.v1.kernel_types.UDPOneToOneRXType;
import com.maxeler.networking.v1.kernel_types.UDPOneToOneTXType;

public class FieldSwapKernel extends FramedKernel {

    private static class TestFrameFormat extends FrameFormat {
        public TestFrameFormat() {
            super(ByteOrder.LITTLE_ENDIAN);

            addField("field_a", dfeUInt(32));
            addField("field_b", dfeUInt(32));
        }
    }

    FieldSwapKernel(KernelParameters parameters) {
        super(parameters);

        TestFrameFormat testFrameFormat = new TestFrameFormat();

        FrameData<TestFrameFormat> frameIn =
            io.frameInput("frameIn", testFrameFormat, new UDPOneToOneRXType());
        FrameData<TestFrameFormat> frameOut =
            new FrameData<TestFrameFormat>(this, testFrameFormat, new UDPOneToOneTXType());

        frameOut["field_a"] <== frameIn["field_b"];
        frameOut["field_b"] <== frameIn["field_a"];
        frameOut.linkfield[UDPOneToOneTXType.SOCKET] <==
            frameIn.linkfield[UDPOneToOneRXType.SOCKET];

        io.frameOutput("frameOut", frameOut);
    }
}
```

The `CH2_SFP1` parameter identifies the physical hardware connection used by this application. The names in quotes match the names of the Kernel input and output streams.

Listing 2: Field swap example Manager (FieldSwapManager.maxj)

```
package fieldswap;

import static com.maxeler.maxcompiler.v2.managers.standard.Manager.UDP;
import static com.maxeler.maxcompiler.v2.managers.standard.Manager.link;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.NetworkConnection;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.UDPConnectionMode;

public class FieldSwapManager {
    public static void main(String[] args) {
        Manager m = new Manager(new FieldSwapEngineParameters(args));

        Kernel k = new FieldSwapKernel(m.makeKernelParameters("fieldSwapKernel"));
        m.setKernel(k);

        m.setIO(link("frameIn",   UDP(NetworkConnection.CH2_SFP1, UDPConnectionMode.OneToOne)),
                link("frameOut", UDP(NetworkConnection.CH2_SFP1, UDPConnectionMode.OneToOne)));
        m.build();
    }
}
```

2

Getting Started

This chapter demonstrates how to use MaxIDE to import and run the Field swap application from *subsubsection 1.5.3* and other networking examples provided with MaxCompiler.

2.1 Launching MaxIDE

To launch MaxIDE, enter the command `maxide` at a command prompt. The MaxIDE welcome page should appear on the screen. If the welcome screen does not appear (for example because it has been disabled in a previous MaxIDE session), you can return to the welcome page by clicking on the *Help* menu at the top and selecting the *Welcome* option.

2.2 Importing the Examples

Clicking "Auto-import MaxCompiler tutorial projects" on the welcome page brings up the import wizard dialog box. The wizard allows you to import the project source code of any MaxCompiler tutorial example or exercise into MaxIDE. To import the field swap example from *subsubsection 1.5.3*, select "maxcompiler-network-tutorial", "examples" and "network_chap01_example1" and click *Finish*.

When the import is complete, MaxIDE switches to the MaxCompiler Project perspective and the field swap example appears in the Project Explorer.

2.3 Building and Running the Examples in MaxIDE

The newly imported example project consists of *CPU Code*, *Engine Code* and *Run Rules* which can be accessed by expanding the project hierarchy in the Project Explorer.

To build and run the field swap example in simulation, expand the *Run Rules* item, right-click on *Simulation* and select the *Run* option. After compilation has finished, the example should execute automatically and write its outputs to the MaxIDE console.

Building a project and running it on a DFE is done in a similar way using the *DFE* Run Rule. Note however that the arguments to the CPU application very likely require adjustment to your specific network setup.

2.4 Configuring Simulation for Networking

The networking examples provided with MaxCompiler can usually be run directly without the need for further configuration. However, the DFE network simulation must be configured manually when creating new networking projects from scratch or if the default settings provided with the examples conflict with the local network configuration.

As pointed out *subsubsection 1.1.2*, the simulation uses Linux TAP devices to act as local network interfaces with a point-to-point connection to one of the simulated DFE network interfaces (SFP1 or SFP2). CPU applications can connect to the simulated DFE through the TAP devices using regular BSD sockets. The hardware equivalent of this setup would be connecting a local NIC to one of the DFE's network interfaces using a crossover cable.

Although the TAP devices are created automatically by the MaxCompiler DFE simulator, they must be enabled by the user by setting an IP address and network mask for each required TAP device. Which TAP devices must be enabled is determined by the network interfaces used in the DFE design.

To set the IP address and netmask for a TAP device, double-click the *Simulation* Run Rule in the Project Explorer. A dialog with Run Rule settings should appear on the screen as shown in *Figure 10*. Tick the “Enable SFP1” or “Enable SFP2” box and enter the desired IP address and network mask.

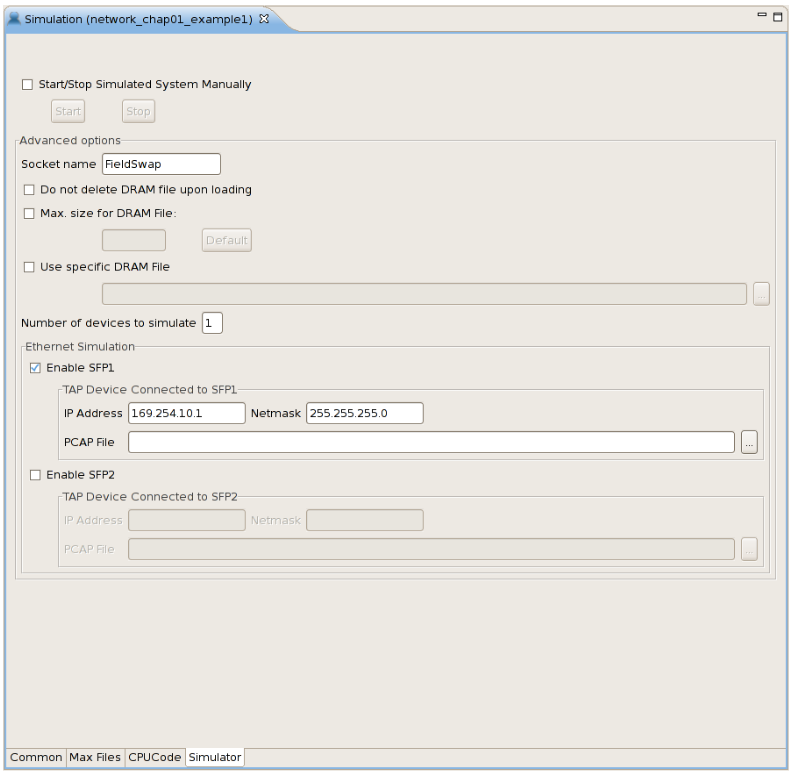
✘ The IP addresses specified in the Simulation Run Rule settings apply to the local TAP devices only. To assign IP addresses to the DFE network interfaces, use the CPU API described in *subsection 4.3*.

Note that the IP address must be chosen to reside on a different subnet from any other real or virtual active connections. It is not possible to enable direct communication between the two simulated DFE network interfaces by specifying IP addresses on the same subnet for the TAP devices.

The Run Rule settings dialog also allows the specification of an optional PCAP file name for each TAP device. If a file name is supplied, all network traffic on the respective TAP device will be written to the file automatically by the DFE simulator. An application such as Wireshark can be used to read and debug the captured network traffic.

2.5 Building and Running the Examples outside MaxIDE

Although highly recommended, MaxIDE is not required for running the MaxCompiler networking examples. The source code for any project imported into MaxIDE is accessible in a directory under your designated MaxIDE workspace directory (typically `$HOME/workspace`). Project directory hierarchies are organized and named identically to the hierarchy of headings in the Project Explorer panel


Simulation (network_chap01_example1)

☐ Start/Stop Simulated System Manually

Advanced options

Socket name: FieldSwap

☐ Do not delete DRAM file upon loading

☐ Max. size for DRAM File:

☐ Use specific DRAM File

Number of devices to simulate: 1

Ethernet Simulation

☑ Enable SFP1

TAP Device Connected to SFP1

IP Address: 169.254.10.1 Netmask: 255.255.255.0

PCAP File:

☐ Enable SFP2

TAP Device Connected to SFP2

IP Address: Netmask:

PCAP File:

Figure 10: Screen shot of the Simulation Run Rule settings

(without spaces). Hence, under each project subdirectory, there are sub-directories named `CPUCode`, `EngineCode`, and `RunRules`.

- The `CPUCode` directory contains C or C++ source files and header files for the project.
- The `EngineCode` directory contains a `src` subdirectory and possibly a `bin` subdirectory.
  - The `bin` subdirectory stores compiled MaxJ class files, if any.
  - The `src` subdirectory has exactly one subdirectory named after the project. This subdirectory contains MaxJ source code files.
- The `RunRules` directory contains a subdirectory named `DFE` a subdirectory named `Simulation`, each containing automatically generated configuration files and Makefiles.

To build a project outside of MaxIDE for a DFE or simulation, navigate to the corresponding `DFE` or `Simulation` directory of the project hierarchy, and invoke the `make` utility using one of the automatically generated Makefiles with an optional target.

- `make` – with no target builds either a simulation model or a DFE configuration `.max` file for the application without running it.
- `make startsim` – starts a simulator if invoked from the `Simulation` directory and there is no simulator already running, but has no effect if invoked in the `DFE` directory or when a simulator is already running.
- `make run` – builds the application if necessary, and then runs it either in a DFE or in simulation, depending on the directory.

  - For DFE runs, DFE hardware is needed.
  - For simulation runs, an already running simulator started by `make startsim` is needed.

- `make stopsim` – invoked from the `DFE` directory has no effect. From the `Simulation` directory, it either stops a simulator if one is running, or causes an error if not.
- `make runsim` – is equivalent to `make startsim run stopsim`

3

Framed Kernels

Network protocols enable interoperability between applications on a wide range of hosts and across a wide range of networks. This requires a rigid set of interfaces and a simple way to communicate both data and metadata to the network and remote host(s). To this end, small chunks of data known as “frames” or “packets” are combined with “headers” from the different network layers before being transmitted.

These headers typically have a fixed or semi-fixed set of “fields” at well-known offsets from the start of the header. This may also extend to user data as well, which may have a fixed or variable length.

The combination of in-band control, fixed-position fields, variable length fields and unpredictable arrival time presents unique challenges to an otherwise static Dataflow model. MaxCompiler provides **Framed Kernels** which greatly simplify dealing with data frames of varying length, and dealing with fields in an intuitive way, regardless of the size of the underlying DFE link.

A Framed Kernel employs input and output streams that may be viewed as generalizations of the streams used in ordinary Kernels, in that they carry words of variable widths apparently in constant time.

This chapter provides the information you need to get started with Framed Kernels. After a short introduction we present the field accumulator example Kernel which subsequently will be used in the remaining sections to illustrate the features of a Framed Kernel.

3.1 Introduction

The Framed Kernel API is available by extending the `FramedKernel` class rather than the usual `Kernel` class. It allows the programmer to create framed inputs and outputs using user-defined formats with the `FrameFormat` class, and DFE link formats with the `FramedLinkType` class.

A Framed Kernel “runs” when a Frame is available on inputs that are “reading”. Once running, each input and output will process either zero or one Frame(s) depending on the state of their control inputs.

Between frames, all stateful elements of the Framed Kernel (except memory elements, such as `mem.rom`, `mem.ram`) are reset. This behavior can be changed on request to maintain state between frames.

3.1.1 Field Concepts

Fields can either be fixed length or variable length. A fixed length field can be any basic (unstructured) MaxCompiler type. All fixed-length fields will be presented co-incident at the start of the frame.

Variable length fields have an “underlying type”, and a granularity. These fields will be presented over a number of cycles in the Kernel.

The frame format can be designed independently of the network communication protocol used for carrying the frames. The same format can be carried by UDP, TCP, or even raw Ethernet depending only on parameterization. MaxCompiler automatically generates hardware using pre-defined DFE link formats for each of these protocols.

3.1.2 Timing Conventions

A crucial benefit of the Framed Kernel API is the timing of fields being presented.

- For Framed Kernels, time passes in discrete ticks. When a frame is transmitted to a Framed Kernel, all of its fixed length fields are instantly available on the first tick. The first element of each variable length field is also available on the first tick.

- Subsequent elements of variable length fields, if any, are delivered in subsequent tick, with no more than one tick needed for each element.

These timing properties apply to frames of any size and format with any number of fields. It may happen that a large frame takes longer to transmit than a small one in real time, but this difference is dealt with by MaxCompiler. It’s common that a field is too long to be carried in one piece on a physical DFE link. In this case, the field is automatically split across words.

3.1.3 Visualizing a Framed Kernel

The timing model presented by the Framed Kernel API is depicted informally in *Figure 11*. The figure represents a situation in which incoming frames with fields a, b, and c are transformed to outgoing frames with fields d and e by some unspecified computation. All fields have a fixed length.

The physical DFE links carrying the frames are 64 bits wide, but the frames are longer. The incoming link delivers field a and part of field b in the first word, with the rest of field b and field c in the second word. The outgoing link carries field d and part of field e first, and the rest of field e second. None of these data partitioning operations is explicitly represented in the Kernel source code. The Kernel is written as if fields a, b, and c are all available simultaneously as mathematical operands, and as if d and e can be transmitted in parallel.

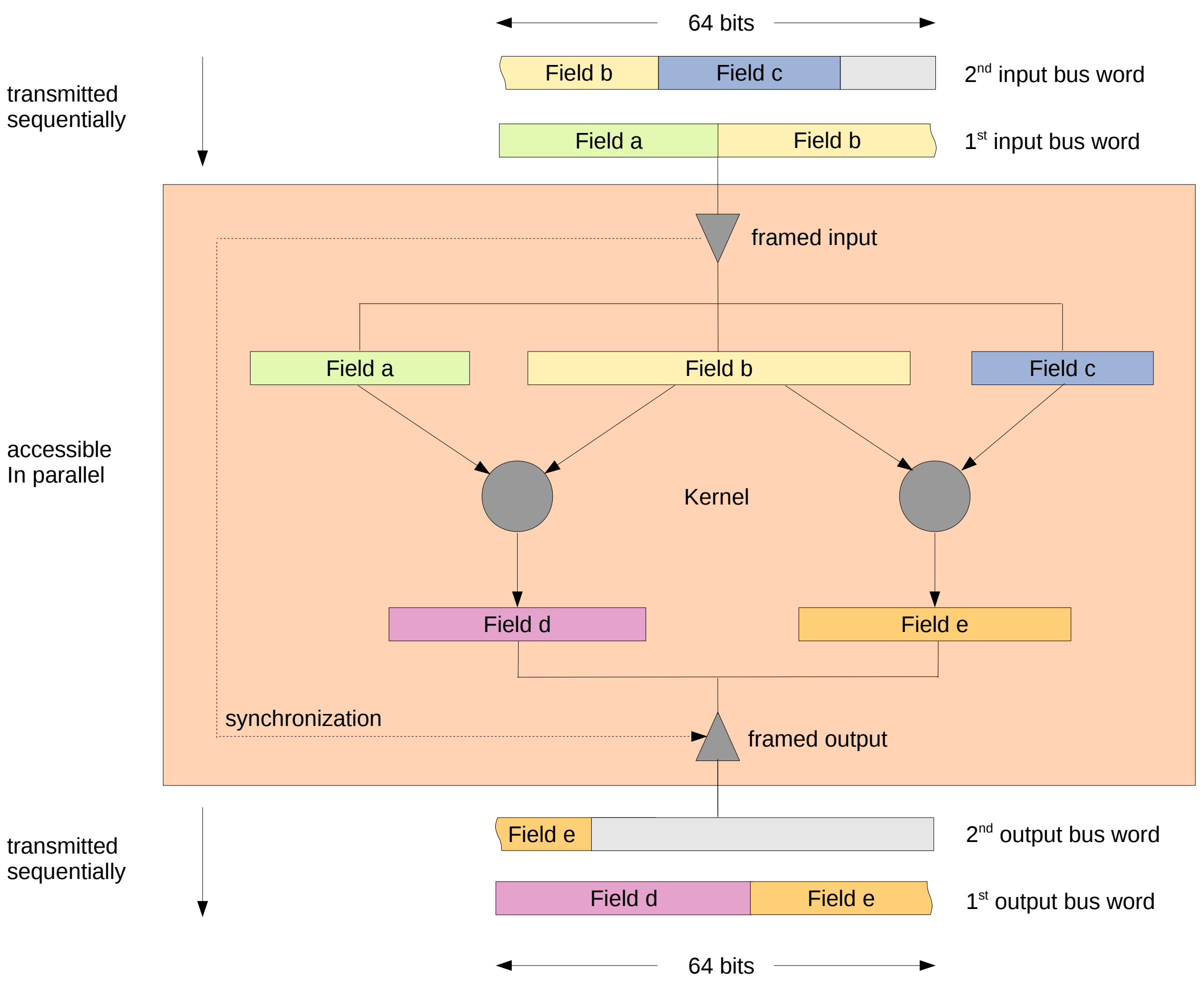


Figure 11: How data carried in pieces by a fixed width link gets together inside a Framed Kernel

3.2 Field Accumulator Example

Before looking at Framed Kernels in more detail we first introduce the field accumulator application which will be used as an example throughout the following sections.

The field accumulator receives frames containing a variable number of integer fields (“items”) via UDP. An additional “numItems” field indicates the actual number of items contained in the frame.

For each received frame, the Kernel calculates the sum of all items contained in that frame as well as the total sum of all items received so far. The two sums are then transmitted back to the receiver. To simplify the Kernel logic we assume that frames always contain an even number of items.

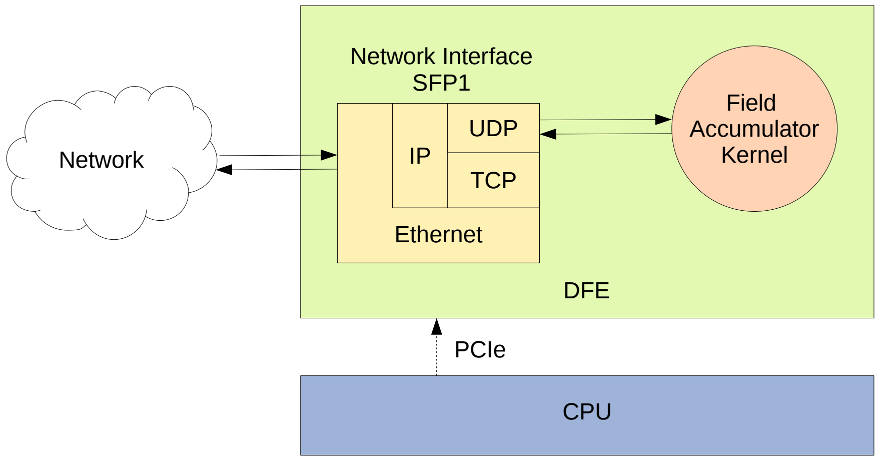


Figure 12: Block diagram of the field accumulator application

Figure 12 depicts a block diagram of the field accumulator application. Only the Kernel shown in listing *Listing 3* and the Manager shown in listing *Listing 4* are written by the developer. A network interface is generated automatically by MaxCompiler. The Manager effectively connects the Kernel to a bi-directional UDP stream provided by the network interface.

The CPU does not participate in the steady state operation of the application and does not feature in the Manager code. However, it invokes the API calls needed to launch and terminate the application, including the source and destination of the UDP stream. This is further explained in *section 4* and *section 5*.

In order to form a self-contained example, the field accumulator CPU code also connects to the local DFE via a conventional BSD socket, sends input datagrams to the DFE and receives output datagrams generated by the DFE. Note that in practice, the DFE would usually communicate to a remote host and not to the local host like in our example.

Listing 3: Field accumulator example Kernel (FieldAccumulatorKernel.maxj)

```
package fieldaccumulator;

import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.Accumulator.Params;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.Reductions;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.networking.v1.framed_kernels.ByteOrder;
import com.maxeler.networking.v1.framed_kernels.FrameData;
import com.maxeler.networking.v1.framed_kernels.FrameFormat;
import com.maxeler.networking.v1.framed_kernels.FramedKernel;
import com.maxeler.networking.v1.kernel_types.UDPOneToOneRXType;
import com.maxeler.networking.v1.kernel_types.UDPOneToOneTXType;

public class FieldAccumulatorKernel extends FramedKernel {
    static class DataIn extends FrameFormat {
        DataIn() {
            super(ByteOrder.LITTLE_ENDIAN);
            addField("numItems", dfeUInt(8));
            addVariableLengthField("items", dfeUInt(8), 1, 255, 2);
        }
    }

    static class DataOut extends FrameFormat {
        DataOut() {
            super(ByteOrder.LITTLE_ENDIAN);
            addField("totalItems", dfeUInt(32));
            addField("sum", dfeUInt(32));
        }
    }

    FieldAccumulatorKernel(KernelParameters parameters) {
        super(parameters);

        FrameData<DataIn> frameIn = io.frameInput("frameIn", new DataIn(), new UDPOneToOneRXType());

        frameIn.setSizeForVariableField("items", frameIn["numItems"]);

        DFEVar cycleCount = control.count.simpleCounter(8);
        DFEVar beyondLastItem = cycleCount > ((frameIn["numItems"] - 1) >> 1);

        DFEVar currentSum = frameIn.getAsVector("items")[0].cast(dfeUInt(32)) + frameIn.getAsVector("items")[1].cast(dfeUInt(32));

        Params sumAccParams =
            Reductions.accumulator.makeAccumulatorConfig(dfeUInt(32)).withEnable(~beyondLastItem);

        DFEVar sum = Reductions.accumulator.makeAccumulator(currentSum, sumAccParams);

        Params itemsAccParams =
            Reductions.accumulator.makeAccumulatorConfig(dfeUInt(32)).withEnable(cycleCount === 0);

        pushResetBetweenFrames(false);
        DFEVar totalItems = Reductions.accumulator.makeAccumulator(
            frameIn["numItems"].cast(dfeUInt(32)),
            itemsAccParams);
        popResetBetweenFrames();

        FrameData<DataOut> frameOut = new FrameData<DataOut>(this, new DataOut(), new UDPOneToOneTXType());

        frameOut["sum"] <== sum;
        frameOut["totalItems"] <== totalItems;
        frameOut.linkfield[UDPOneToOneTXType.SOCKET] <==
            frameIn.linkfield[UDPOneToOneRXType.SOCKET];

        io.frameOutput("frameOut", frameOut, constant.var(true), beyondLastItem);
    }
}
```

Listing 4: Field accumulator example Manager (FieldAccumulatorManager.maxj)

```
package fieldaccumulator;

import static com.maxeler.maxcompiler.v2.managers.standard.Manager.UDP;
import static com.maxeler.maxcompiler.v2.managers.standard.Manager.link;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.NetworkConnection;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.UDPConnectionMode;

public class FieldAccumulatorManager {
    public static void main(String[] args) {
        Manager m = new Manager(new FieldAccumulatorEngineParameters(args));

        Kernel k = new FieldAccumulatorKernel(m.makeKernelParameters("fieldAccumulatorKernel"));
        m.setKernel(k);

        m.setIO(link("frameIn",   UDP(NetworkConnection.CH2_SFP1, UDPConnectionMode.OneToOne)),
                link("frameOut", UDP(NetworkConnection.CH2_SFP1, UDPConnectionMode.OneToOne)));
        m.build();
    }
}
```

3.3 Frame Format

A flexible style of data description enables Framed Kernel streams to cater for many applications. The data structure is described as a sequence of arbitrarily many fields using the `FramedFormat` class. Each field is specified by a string name and a type, the type determining the size and format of the field.

The field accumulator Kernel uses two `FramedFormats`. `DataIn` describes the input frames received from a remote peer consisting of a fixed length integer field `numItems` and a variable length integer field `items`:

```
static class DataIn extends FrameFormat {
    DataIn() {
        super(ByteOrder.LITTLE_ENDIAN);
        addField("numItems", dfeUInt(8));
        addVariableLengthField("items", dfeUInt(8), 1, 255, 2);
    }
}
```

`DataOut` specifies the output frames that are transmitted back to the remote peer and contains the two fixed length integer fields `totalItems` and `sum`:

```
static class DataOut extends FrameFormat {
    DataOut() {
        super(ByteOrder.LITTLE_ENDIAN);
        addField("totalItems", dfeUInt(32));
        addField("sum", dfeUInt(32));
    }
}
```

The characteristics of fixed and variable length fields are further described in the next two sections.

3.3.1 Fixed Length Fields

The following method is used for defining the fixed length fields in a frame format:

***void** addField(**String** name, DFEType type)*

This method is invoked once for each field, from within the constructor method of a class that extends the `FrameFormat` class, as shown by the field accumulator code in the previous section.

The first field declared is the first to be transmitted on the stream, and the rest follow in order.

To design your application for compatibility with an existing protocol message format, list the fields in the order the protocol requires.

3.3.2 Variable Length Fields

For more general types of data, the following method expresses a field of variable length, for example a character string or a sequence of numbers.

***void** addVariableLengthField(**String** name, DFEType type, **int** minElements, **int** maxElements)*

The `minElements` and `maxElements` parameters are the minimum and maximum numbers of elements ever allowed in a field of this type. They must be non-negative, and should be no more general than necessary due to hardware resource overheads incurred.

The `addVariableLengthField` method is invoked similarly to `addField` (*subsubsection 3.3.1*), as part of a sequence within the constructor of a class extending `FrameFormat`. Fixed and variable length fields may be freely mixed.

Note that our field accumulator example uses an advanced version of `addVariableLengthField` with an additional parameter to specify a field access granularity. The concept of granularities is further explained in *subsubsection 3.6.3*.

The specification of a variable length field is not complete without further information. Although the minimum and maximum possible lengths are known, the actual length of a given field at run time also needs to be communicated somehow. The following method helps in this regard:

void *setSizeForVariableField(**String** fieldName, DFEVar numElements)*

This method is called from the Kernel and associates a hardware variable containing a length with a variable length field. The variable must be an integer type of sufficient width to represent the number of elements. In the field accumulator example this variable is derived from the fixed length field `numItems`:

```
36        frameIn.setSizeForVariableField("items", frameIn["numItems"]);
```

In theory the variable could be calculated in any way at all by a Kernel, subject only to the constraint that it contain the actual correct number of elements in the field.

The `setSizeForVariableField` method is applicable to objects of type `FrameData<`*T*`>`, where *T* is a class extending `FrameFormat` (e.g., `DataIn` in our example). A useful idiom for defining the length of a field as another field in the same frame is the following,

```
f.setSizeForVariableField("items", f["numItems"]);
```

where *f* is an object instantiated previously by one of these methods

```
FrameData<T> f = io.frameInput(···);
FrameData<T> f = new FrameData<T> (···);
```

which are explained in *subsection 3.5*. The field specifying the length must appear before the related variable length field in the frame. Note the usage of square brackets and a field identifier to retrieve a DFEVar for a field in a frame.

3.4 Stream Format

As we have seen in the previous section, frame formats purely define the structure of user data without committing to an underlying network protocol and any associated metadata. Only when instantiating network streams by creating frame IOs (see *subsection 3.5*) are the frame formats tied to a stream format by specifying protocol-dependent DFE Link type.

Link types define the metadata fields carried by a stream alongside the user data. While some of these fields need to be handled explicitly by the developer (see the sections listed in *Table 1*), Framed Kernels automatically interpret the control metadata needed to transmit variable-length network packets over a fixed-width stream. For this purpose each packet is split across multiple fixed width words that can go through a stream, along with control metadata to signal the beginning and end of a packet.

Protocol	Type	Specification
TCP	TCPType	*subsection 6.1*
UDP	UDPOneToManyRXType	*subsubsection 5.2.2*
	UDPOneToManyTXType	*subsubsection 5.2.1*
	UDPOneToOneRXType	*subsubsection 5.1.2*
	UDPOneToOneTXType	*subsubsection 5.1.1*
Ethernet	EthernetRXType	*subsection 8.1*
	EthernetTXType	*subsection 8.2*

Table 1: The choice of a DFE link type to carry framed data is orthogonal to the format.

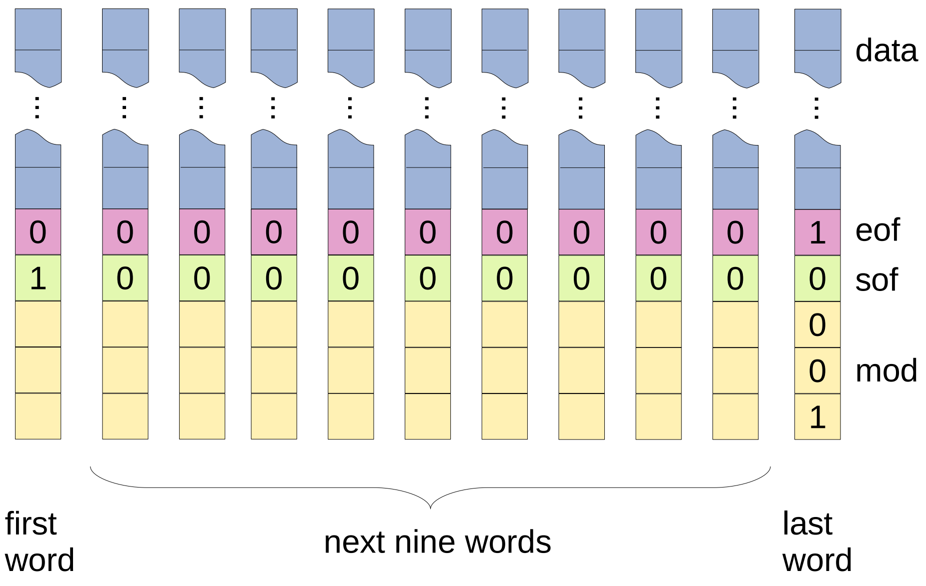


Figure 13: Control metadata in the sequence of 11 words needed for an 84 byte packet

The following fields are common to all MaxCompiler network streams and are handled automatically by Framed Kernels:

- **data** contains eight bytes of payload data, unless EOF is true, in which case it may contain fewer than eight but at least one byte. If it contains fewer than eight bytes, they should be located contiguously starting from bit position zero. The data field never contains data from more than one packet.
- **mod** is ignored when EOF is false. When EOF is true, mod contains the number of bytes of payload data in the data field unless there are eight bytes, in which case mod contains zero.
- **EOF** is true only when the data field contains the last byte of the current packet.
- **SOF** is true only when the data field contains the first bytes of the current packet.

Figure 13 shows an example of a variable length packet transmitted over a fixed-width stream using the control metadata fields introduced above. A packet of 84 bytes takes 11 cycles to be transmitted, with eight bytes of frame data transmitted on each of the first ten cycles, and four on the eleventh. In the first word, SOF is 1, EOF is 0. In the next nine words, SOF is 0 and EOF is 0. In the eleventh word, SOF is 0, EOF is 1 and mod is 4. The last four bytes in the data field of the eleventh word are unoccupied.

3.5 IOs

The following two sections show how to instantiate network streams carrying framed data. Different conventions apply to input and output streams.

3.5.1 Input Streams

This method instantiates a stream to carry frames of the given format.

```
public FrameData<FrameFormatT> io.frameInput(
    String name,
    FrameFormatT type,
    FramedLinkType linkType)
```

The `name` parameter is the name used by the Manager or CPU code to refer to the stream if necessary. The `type` parameter is an instance of a class extending FrameFormat, and the link type is an instance of one of those listed in *Table 1*. In our example we use the following line to add an input called “frameIn” carrying frames in the `DataIn` format over UDP:

```
        FrameData<DataIn> frameIn = io.frameInput("frameIn", new DataIn(), new UDPOneToOneRXType());
```

The `io.frameInput()` method returns a `FrameData` object which is used to access the frames received on the input. Individual fields can be retrieved through the index operator `[]`. In the field accumulator example we use the `numItems` field and a cycle counter to keep track of whether we are beyond the last item in a frame:

```
        DFEVar cycleCount = control.count.simpleCounter(8);
        DFEVar beyondLastItem = cycleCount > ((frameIn["numItems"] - 1) >> 1);
```

3.5.2 Output Streams

For output streams, the following method is used to transmit a frame on the stream of the given name.

```
public void io.frameOutput(
    String name,
    FrameData<FrameFormatT> frameData
```

As for frame input streams, the `name` parameter is the name used by the Manager or CPU code to identify the output stream.

The parameter `dataOut` is a `FrameData` instance parameterized with a user-defined `FrameFormat`. In our field accumulator example we use `DataOut` from *subsection 3.3* as the frame format and UDP as the underlying DFE link type (see *Table 1*) when instantiating the output `FrameData`:

```
        FrameData<DataOut> frameOut = new FrameData<DataOut>(this, new DataOut(), new UDPOneToOneTXType());
```

This is an example of usage of the FrameData constructor

```
public FrameData(FramedKernel kernel, FrameFormatT frameFormat, FramedLinkType framedLinkType)
```

The following lines from the field accumulator Kernel show how to assign values to the fields in `frameOut` (i.e., an instance of the `FrameData` class):

```
        frameOut["sum"] <== sum;
        frameOut["totalItems"]  <== totalItems;
```

That is, a field is identified by the index operator `[]` and the field name in quotes, and assigned the value of an expression using the connection operator <==.

DFE Link format specific metadata is accessible in the same way through the `linkfield` member of `FrameData`. Our field accumulator example uses `linkfield` to set the destination UDP socket to be the same as the source socket, which effectively transmits the Kernel output back to the remote peer:

```
        frameOut. linkfield [UDPOneToOneTXType.SOCKET] <==
            frameIn. linkfield  [UDPOneToOneRXType.SOCKET];
```

3.6 Advanced Features

Several additional features described in this section can make Framed Kernels more convenient or efficient in some applications. These include controlled inputs and outputs, persistent state capabilities, and adjustable granularity for variable length fields.

3.6.1 Controlled Inputs and Outputs

Controlled inputs are used when a Kernel needs to do some work between frames that precludes accepting a new frame on every tick. By disabling the input, the Kernel may continue running while receipt of the next incoming frame, if any, is postponed.

A controlled input requires a Boolean *DFEVar* to enable or disable it, and another Boolean *DFEVar* to indicate whether the enabling variable should be heeded or ignored. The `frameInput` method mentioned in *subsubsection 3.5.1* lets you declare a controlled input by passing these variables as two optional parameters.

```
public FrameData<FrameFormatT> io.frameInput(
    String name,
    FrameFormatT type,
    FramedLinkType linkType,
    DFEVar enableDecision,
    DFEVar enableDecisionReady)
```

The `enableDecision` parameter is used to enable or disable data flowing through the input. `enableDecision` is only sampled when the `enableDecisionReady` parameter is set to `true` and ignored otherwise. Note that `enableDecisionReady` must be set to `true` for every frame eventually and once set, changing it has no further effect.

> ✷ The `enableDecisionReady` parameter should not be used for permanently disabling a stream, because doing so could stall the Kernel. See *subsubsection 3.6.1* for recommendations regarding controlled outputs.

Table 2 shows an example where an input is enabled in tick 4. `enableDecision` is ignored in all ticks but tick 4, when `enableDecisionReady` is asserted. Similarly, in the example in *Table 3* an input is disabled in tick 6.

Tick	0	1	2	3	4	5	6	7	8	9
enableDecision	x	x	x	x	1	x	x	x	x	x
enableDecisionReady	0	0	0	0	1	0	0	0	0	0

Table 2: An input is enabled in tick 4.

Tick	0	1	2	3	4	5	6	7	8	9
enableDecision	x	x	x	x	x	x	0	x	x	x
enableDecisionReady	0	0	0	0	0	0	1	0	0	0

Table 3: An input is disabled in tick 6.

Controlled outputs are useful when an output is to be produced on some ticks but not all. A controlled output is created by using the `enableDecision` and `enableDecisionReady` optional parameters in the `frameOutput` method as shown below.

```
public <FrameFormatT extends FrameFormat> void io.frameOutput(
    String name,
    FrameData<FrameFormatT> frameData,
    DFEVar enableDecision,
    DFEVar enableDecisionReady)
```

The `enableDecision` and `enableDecisionReady` parameters are interpreted the same way as for controlled inputs.

In our field accumulator example we add an output as follows, indicating that the frame is ready to be sent when we have processed the last data item:

```
        io.frameOutput("frameOut", frameOut, constant.var(true), beyondLastItem);
```

3.6.2 Persistent State

Normally a Framed Kernel maintains no persistent state between frames, but this behavior can be overridden by bracketing the declarations of persistent variables with these method invocations.

```
void pushResetBetweenFrames(boolean enableReset)
void popResetBetweenFrames()
```

The `enableReset` parameter specifies whether the state should be reset between frames. If it is false, then variables retain their states between frames.

In our field accumulator example we want to ensure that the accumulator used to sum up the total number of items received retains its value across frames. This is done as follows:

```
        Params itemsAccParams =
            Reductions.accumulator.makeAccumulatorConfig(dfeUInt(32)).withEnable(cycleCount === 0);

        pushResetBetweenFrames(false);
        DFEVar totalItems = Reductions.accumulator.makeAccumulator(
            frameIn["numItems"].cast(dfeUInt(32)),
            itemsAccParams);
        popResetBetweenFrames();
```

3.6.3 Granularity

Normally each element of a variable length field in a frame takes one tick to be received. When the elements are small, it may save time to receive more than one of them in each tick. The number of elements to receive in each tick can be specified by the optional `accessGranularity` parameter to the `FrameFormat` `addVariableLengthField` method:

void addVariableLengthField(String name, DFEType type, int minElements, int maxElements, int accessGranularity)

When a granularity greater than one is chosen, the additional field elements are accessible through the following method of the `FrameData` class:

DFEVector<DFEVar> getAsVector(String fieldName)

The size of the returned `DFEVector` is equal to the granularity. However, in the last tick of a frame, the `DFEVector` will not be completely filled if the number of field elements in the frame is not divisible by the chosen granularity.

As shown in *subsection 3.3*, the field accumulator Kernel uses a granularity of two for the `items` field:

```
    static class DataIn extends FrameFormat {
        DataIn() {
            super(ByteOrder.LITTLE_ENDIAN);
            addField("numItems", dfeUInt(8));
            addVariableLengthField("items", dfeUInt(8), 1, 255, 2);
        }
    }
```

Consequently, we need to build the sum two items on each tick which we pass to an accumulator. The accumulator keeps track of the sum of items in the current frame and is automatically reset between frames:

```
        DFEVar currentSum = frameIn.getAsVector("items")[0].cast(dfeUInt(32)) + frameIn.getAsVector("items")[1].cast(dfeUInt(32));

        Params sumAccParams =
            Reductions.accumulator.makeAccumulatorConfig(dfeUInt(32)).withEnable(~beyondLastItem);

        DFEVar sum = Reductions.accumulator.makeAccumulator(currentSum, sumAccParams);
```

4

IP Connectivity

This chapter shows how a DFE is integrated into a networking CPU application with the aid of the SLiC API. The field accumulator example from *subsection 3.2* is used to illustrate the steps required to set up a DFE and perform basic network layer configuration. Configuring the network layer is a prerequisite for connecting a DFE to a remote peer which is explained in *section 5* for UDP and *section 6* for TCP.

4.1 The Role of SLiC

After creating a Kernel and Manager configuration using MaxCompiler, the next step in the development of a networking dataflow implementation is integrating the CPU application with SLiC. SLiC is the C API for running the DFE and configuring various aspects of it, including the Maxeler networking infrastructure. The steps a CPU networking application will typically need to perform are:

- Open a connection to a DFE or simulated system.
- Configure the dataflow engine.
- Set up the networking infrastructure and connecting to one or several remote peers. Once the network connections are established, the CPU will often not participate in the steady state operation of the DFE in order to maximize performance.

- After the application-specific operations on the DFE are completed, close any previously established network connections and the DFE.

In the next two sections we will look at how the CPU code for the field accumulator example from *subsection 3.2* performs the steps listed above. The full CPU code is shown in *Listing 9* and *Listing 10*.

> In networking applications, engines are implicitly reset when they are allocated. Subsequently, SLiC actions do not automatically reset engines when they are executed.

4.2 General SLiC Setup

The field accumulator example first loads the `.max` file created by the build process using the following line:

```
    max_file_t *maxfile = FieldAccumulatorUDPChap04_init();
```

Next, we call `max_load()` to open and configure the dataflow engine:

```
   max_engine_t * engine = max_load(maxfile, "*");
```

In our example, the argument “*” of will direct `max_load()` to return the appropriate hardware or simulation engine name for the profile currently used to run the application.

We are now ready to configure the Maxeler networking infrastructure and start the actual application-specific processing. The configuration of the network layer components is explained in the following section.

Once the field accumulator is done processing, we simply close the dataflow engine and free up the resources used by the `.max` file:

```
   max_unload(engine);
   max_file_free(maxfile);
```

Listing 5: TCP Field accumulator example CPU code (FieldAccumulatorCpuCode.c)

```
#define _GNU_SOURCE

#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <MaxSLiCInterface.h>
#include "FieldAccumulatorUDPChap04.h"


typedef struct {
    uint32_t total_items ;
    uint32_t sum;
} __attribute__ ((__packed__)) frame_t;


static int create_cpu_udp_socket(struct in_addr *local_ip, struct in_addr *remote_ip, int port) {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);

    struct sockaddr_in cpu;
    memset(&cpu, 0, sizeof(cpu));
    cpu.sin_family = AF_INET;
    cpu.sin_port = htons(port);

    cpu.sin_addr = *local_ip ;
    bind(sock, (struct sockaddr *)&cpu, sizeof(cpu));

    cpu.sin_addr = *remote_ip;
    connect(sock, (const struct sockaddr*) &cpu, sizeof(cpu));

    return sock;
}


static void exchangeItems(int sock, const uint8_t* items, uint8_t num_items) {
    static uint32_t total_items = 0;

    uint8_t data_to_send[1 + UINT8_MAX];
    data_to_send[0] = num_items;
    memcpy(data_to_send + 1, items, num_items);

    send(sock, &data_to_send, num_items + 1, 0);

    frame_t data_received;

    recv(sock, &data_received, sizeof(data_received), 0);

    printf ("Received: total_items = %u, sum = %u\n", data_received.total_items, data_received.sum);

    total_items += num_items;

    uint32_t sum = 0;
    for (int i = 0; i < num_items; i++)
        sum += items[i];

    if (data_received.total_items != total_items || data_received.sum != sum) {
        printf ("Error! Expected: total_items = %u, sum = %u\n", total_items, sum);
        exit (1);
    }
}
```

Listing 6: TCP Field accumulator example CPU code (FieldAccumulatorCpuCode.c)

```
int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf ("Usage: %s <dfe_ip> <cpu_ip> <netmask>\n", argv[0]);
        return 1;
    }

    struct in_addr dfe_ip;
    inet_aton(argv[1], &dfe_ip);
    struct in_addr cpu_ip;
    inet_aton(argv[2], &cpu_ip);
    struct in_addr netmask;
    inet_aton(argv[3], &netmask);
    const int port = 5007;

    max_file_t *maxfile = FieldAccumulatorUDPChap04_init();
    max_engine_t * engine = max_load(maxfile, "*");

    max_ip_config(engine, MAX_NET_CONNECTION_CH2_SFP1, &dfe_ip, &netmask);
    max_udp_socket_t *dfe_socket = max_udp_create_socket(engine, "udp_ch2_sfp1");
    max_udp_bind(dfe_socket, port);
    max_udp_connect(dfe_socket, &cpu_ip, port);

    int cpu_socket = create_cpu_udp_socket(&cpu_ip, &dfe_ip, port);

    uint8_t items1[] = { 2, 5, 8, 1 };
    exchangeItems(cpu_socket, items1, sizeof(items1));

    uint8_t items2[] = { 50, 2 };
    exchangeItems(cpu_socket, items2, sizeof(items2));

    uint8_t items3[] = { 1, 2, 3, 4, 5, 6 };
    exchangeItems(cpu_socket, items3, sizeof(items3));

    close(cpu_socket);

    max_udp_close(dfe_socket);

    max_unload(engine);
    max_file_free(maxfile);

    return 0;
}
```

4.3 Network Layer Setup

For many applications it is sufficient to just call `max_ip_config()` to perform the minimally required network layer setup.

```
void max_ip_config(
    max_engine_t           *engine,
    max_net_connection_t connection,
    const struct in_addr *ip,
    const struct in_addr *netmask);
```

This function associates a physical network connection of the DFE with an IP address and network mask and brings up the Maxeler networking infrastructure. It must be called prior to using any other networking-related parts of the SLiC API. Like many other functions performing networking-related configuration, `max_ip_config()` may only be called once, during the setup phase of an application.

```
max_ip_config(engine, MAX_NET_CONNECTION_CH2_SFP1, &dfe_ip, &netmask);
```

The basic network layer setup shown in this section is common to all networking applications, independent of whether their kernels use UDP, TCP or raw Ethernet as a DFE link type. All link type specific parts of the SLiC API are introduced in *section 5* for UDP, *section 6* for TCP and *section 8* for raw Ethernet. Advanced network layer options which might be required by specialized applications are introduced in *section 9*.

5

UDP Packet Processing

This chapter introduces UDP processing concepts and techniques in the context of the Maxeler network infrastructure. The first part of the chapter shows details on the different UDP stream formats available in MaxCompiler. Only stream fields requiring manual handling by the developer are discussed here. An overview of the automatically handled fields common to all MaxCompiler network streams is provided in *subsection 3.4*.

The UDP infrastructure offers two distinct types of streams supporting different use cases:

- **One-to-one** streams exchange all UDP packets with the same remote peer. The remote peer is set by the CPU code during initialization. One-to-one streams allow for simpler DFE designs but offer less flexibility.
- **One-to-many** streams can exchange UDP packets with any remote peer. The remote peer is dynamically set by the DFE for each UDP packet. One-to-many streams offer greater flexibility but may require more complex DFE designs.

The second part of this chapter shows how to configure UDP streams from the CPU via the SLiC API, using the field accumulator application introduced in *subsection 3.2* as an example.

Field	Offset (bits)	Width (bits)
data	0	64
mod	64	3
eof	67	1
sof	68	1
socket	69	8

Table 4: UDP one-to-one mode transmit stream format

Field	Offset (bits)	Width (bits)
data	0	64
mod	64	3
eof	67	1
sof	68	1
checksum_bad	69	1
socket	70	8

Table 5: UDP one-to-one mode receive stream format

5.1 One-to-one mode UDP streams

In one-to-one mode, all outgoing UDP packets are sent to and received from the same remote peer set by the CPU code. Applications using this mode do not specify destination addresses in the data streams, because the remote peer's address parameters are stored as constants when the application initiates the communication.

5.1.1 One-to-one transmit format

The format used by streams carrying UDP data in the direction toward the network in one-to-one mode is shown in *Table 4* (MaxJ class `UDPOneToOneTXType`). Applications transmitting data on a stream in this format must set the following metadata field:

- **socket** is a number between 0 and 15 identifying the socket from which the packet is sent to the remote peer. Applications can obtain the socket number for any open connection through an API call (*subsubsection 5.3.1*). The socket number must only be specified at the start of a packet, when SOF is true. It is ignored at all other times. Specifying an invalid socket number causes the packet not to be transmitted. It needs to be specified only at the start per packet, when SOF is true. It is ignored at all other times.

5.1.2 One-to-one receive format

The format used by streams carrying UDP data in the direction from the network in one-to-one mode is shown in *Table 5* (MaxJ class `UDPOneToOneRXType`). Most of the fields are similar to the transmit format described in *subsubsection 5.1.1*, but there is an additional checksum field. Applications receiving data from one of these streams should interpret the fields as follows.

- **checksum_bad** is true only if the current packet is found to contain an error based on the checksum, and manual checksum verification is enabled. If automatic checksum verification is used

Field	Offset (bits)	Width (bits)
data	0	64
mod	64	3
eof	67	1
sof	68	1
dst_mac	69	48
dst_ip	117	32
dst_port	149	16
socket	165	8

Table 6: UDP one-to-many mode transmit stream format

(the default), this field is always false as bad packets are dropped before reaching the Kernel. The checksum_bad field is defined whenever EOF is true.

- **socket** contains a number between 0 and 15 identifying the socket on which the packet was received, which is useful if the local application is communicating with the remote peer on more than one port. Applications can obtain the socket number for any open connection through an CPU API call (*subsubsection 5.3.1*). Although the socket number may vary between packets, it is always the same throughout any given packet.

5.2 One-to-many Mode UDP Streams

In the one-to-many mode, UDP streams carry similar information to the one-to-one mode streams, and also include the destination address parameters for the remote peer. The one-to-many mode stream formats are described in greater detail below.

5.2.1 One-to-many Transmit Format

The format used for multi-mode UDP streams carrying data outward to the network is depicted in *Table 6* (MaxJ class `UDPOneToManyTXType`). The socket field has identical semantics to its counterpart in *subsubsection 5.1.1*. The destination address fields are for the MAC address, IP address and port of the remote peer to which the packet is sent and must be specified in network byte order.

Although the address fields can differ between packets, they are constant within each packet. Therefore, the sender need not specify them except during the first word of each packet, when SOF is true. They are ignored when SOF is false.

5.2.2 One-to-many Receive Format

The format used for streams carrying incoming UDP traffic from the network in one-to-many mode is depicted in *Table 7* (MaxJ class `UDPOneToManyRXType`). The checksum_bad and socket fields have identical interpretations to their counterparts in *subsubsection 5.1.2*. The source address fields are similar to those of *subsubsection 5.2.1*, except that they are always valid (not just during SOF).

5.3 UDP CPU API Summary

The API functions sufficient for most applications using UDP are listed in this section for reference. Note that all sockets needed by an application during its entire run must be created before any are used. This

Field	Offset (bits)	Width (bits)
data	0	64
mod	64	3
eof	67	1
sof	68	1
src_mac	69	48
src_ip	117	32
src_port	149	16
checksum_bad	165	1
socket	166	8

Table 7: UDP one-to-many mode receive stream format

limitation is necessary because certain initialization functions performed automatically when a socket is first used are not exposed by the API.

In the following sections, some of the fundamental API functions are illustrated by the field accumulator example from *subsection 3.2*. We have seen in *section 4* how the basic SLiC and network layer setup is performed. What remains to be done is the configuration of the single UDP one-to-one stream used in the example. The CPU code for the field accumulator is shown in *Listing 9* and *Listing 10*.

5.3.1 Socket Management

A data structure known as a **socket** is used to track the state of the UDP protocol handling hardware. A reference to a socket in the form of a pointer of type `max_udp_socket_t*` is needed for most API calls. A pointer of this type can be generated by the following function, which also performs some hardware initialization.

*max_udp_socket_t∗ max_udp_create_socket (max_engine_t ∗engine, **const char** ∗stream_name)*

The `engine` parameter is an engine handle returned by a prior call to `max_load`, and the stream parameter is the name of the relevant UDP stream specified in the Manager.

Getting a socket number When a socket is created by the function above, a socket number is assigned to it automatically. The socket number appears in the socket field of every stream word passing through the connection, as shown in *Table 4* through *Table 7*, in case it is needed by the application to distinguish among multiple connections carried by the same stream. To obtain the number assigned to a given socket, use this function.

uint16_t max_udp_get_socket_number (max_udp_socket_t∗ udp_socket)

The socket number returned by `max_udp_get_socket` can be passed to a Kernel, for example, using a scalar input.

Setting a socket number As an alternative to letting the socket number be chosen by the run time system, a socket can be created with a socket number between 0 and 15 using the following function.

max_udp_socket_t∗ max_udp_create_socket_with_number (
max_engine_t∗ engine,
***const char**∗ stream_name,*
uint16_t socket_number)

Listing 7: UDP Field accumulator example CPU code (FieldAccumulatorCpuCode.c)

```
#define _GNU_SOURCE

#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <MaxSLiCInterface.h>
#include "FieldAccumulatorUDPChap05.h"


typedef struct {
      uint32_t total_items ;
      uint32_t sum;
} __attribute__ ((__packed__)) frame_t;


static int create_cpu_udp_socket(struct in_addr *local_ip, struct in_addr *remote_ip, int port) {
      int sock = socket(AF_INET, SOCK_DGRAM, 0);

      struct sockaddr_in cpu;
      memset(&cpu, 0, sizeof(cpu));
      cpu.sin_family = AF_INET;
      cpu.sin_port = htons(port);

      cpu.sin_addr = *local_ip ;
      bind(sock, (struct sockaddr *)&cpu, sizeof(cpu));

      cpu.sin_addr = *remote_ip;
      connect(sock, (const struct sockaddr*) &cpu, sizeof(cpu));

      return sock;
}


static void exchangeItems(int sock, const uint8_t* items, uint8_t num_items) {
      static uint32_t total_items = 0;

      uint8_t data_to_send[1 + UINT8_MAX];
      data_to_send[0] = num_items;
      memcpy(data_to_send + 1, items, num_items);

      send(sock, &data_to_send, num_items + 1, 0);

      frame_t data_received;

      recv(sock, &data_received, sizeof(data_received), 0);

      printf ("Received: total_items = %u, sum = %u\n", data_received.total_items, data_received.sum);

      total_items += num_items;

      uint32_t sum = 0;
      for (int i = 0; i < num_items; i++)
            sum += items[i];

      if (data_received.total_items != total_items || data_received.sum != sum) {
            printf ("Error! Expected: total_items = %u, sum = %u\n", total_items, sum);
            exit (1);
      }
}
```

Listing 8: UDP Field accumulator example CPU code (FieldAccumulatorCpuCode.c)

```c
int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf ("Usage: %s <dfe_ip> <cpu_ip> <netmask>\n", argv[0]);
        return 1;
    }

    struct in_addr dfe_ip;
    inet_aton(argv[1], &dfe_ip);
    struct in_addr cpu_ip;
    inet_aton(argv[2], &cpu_ip);
    struct in_addr netmask;
    inet_aton(argv[3], &netmask);
    const int port = 5007;

    max_file_t *maxfile = FieldAccumulatorUDPChap05_init();
    max_engine_t * engine = max_load(maxfile, "*");

    max_ip_config(engine, MAX_NET_CONNECTION_CH2_SFP1, &dfe_ip, &netmask);

    max_udp_socket_t *dfe_socket = max_udp_create_socket(engine, "udp_ch2_sfp1");
    max_udp_bind(dfe_socket, port);
    max_udp_connect(dfe_socket, &cpu_ip, port);

    int cpu_socket = create_cpu_udp_socket(&cpu_ip, &dfe_ip, port);

    uint8_t items1[] = { 2, 5, 8, 1 };
    exchangeItems(cpu_socket, items1, sizeof(items1));

    uint8_t items2[] = { 50, 2 };
    exchangeItems(cpu_socket, items2, sizeof(items2));

    uint8_t items3[] = { 1, 2, 3, 4, 5, 6 };
    exchangeItems(cpu_socket, items3, sizeof(items3));

    close(cpu_socket);

    max_udp_close(dfe_socket);

    max_unload(engine);
    max_file_free(maxfile);

    return 0;
}
```

Closing a socket The following function stops a socket from receiving or transmitting.

max_udp_socket_t max_udp_close (max_udp_socket_t* socket)*

It is good practice to close a socket before terminating the application to avoid possible issues with spurious transmissions.

In the field accumulator example, a socket is created for the UDP stream on network connection SFP1, identified by the automatically generated stream name `"udp_ch2_sfp1"`:

```
max_udp_socket_t *dfe_socket = max_udp_create_socket(engine, "udp_ch2_sfp1");
```

After processing all data but before closing the dataflow engine, the socket is closed again:

```
max_udp_close(dfe_socket);
```

5.3.2 Receiving

After all necessary sockets have been created, the following function can be called to enable a socket to receive packets sent to the specified port.

void *max_udp_bind (max_udp_socket_t* socket, uint16_t local_port)*

Note that no local IP address can be specified here and the address previously set by `max_ip_config` (see *section 4*) will be used.

The field accumulator example binds its socket to a user-supplied local port:

```
max_udp_bind(dfe_socket, port);
```

5.3.3 Sending

If a UDP stream is initialized in one-to-many mode in the Manager, the destination addresses for outgoing packets are set in the Kernel code. However, one-to-one mode configurations require the following function call to set the constant destination address for all packets.

void *max_udp_connect (max_udp_socket_t* socket,* ***const struct*** *in_addr *remote_ip, uint16_t remote_port)*

This function should be called only after all sockets have been created.

In the field accumulator example, which uses a one-to-one stream, we connect the DFE back to the node running the CPU code:

```
max_udp_connect(dfe_socket, &cpu_ip, port);
```

5.3.4 Multicast

The multicast feature of UDP is supported by the following functions. Calling this function enables a socket to receive packets sent to the IP address and port given by the parameters.

void *max_udp_bind_ip (max_udp_socket_t* socket,* ***const struct*** *in_addr* local_ip, uint16_t local_port)*

The IP address passed to `max_udp_bind_ip` must be that of an existing multicast group previously joined by a call to `max_ip_multicast_join_group` (*section 9*).

The following function can be used to define the time to live for outgoing packets.

void *max_udp_set_ttl (max_udp_socket_t ∗socket, uint8_t ttl)*

The `ttl` parameter specifies the number hops an outgoing packet may take before being automatically dropped.

6

TCP Segment Processing

This chapter introduces TCP processing concepts and techniques in the context of the Maxeler network infrastructure. Following a summary of the TCP stream format, this chapter shows how to configure TCP streams from the CPU via the SLiC API, using a version of the field accumulator application from *subsection 3.2* adapted to TCP as an example.

6.1 TCP Stream Format

TCP data is conveyed in blocks of arbitrary size known as **segments**. While the data is conventionally described as being a continuous stream, in practice MaxCompiler DFE designs process the data in segments, treating each segment as a Framed Kernel frame. In general, the segment sizes are determined automatically by the networking hardware with no relation to the meaning of the information they contain.

The TCP stream format is shown in *Table 8* (MaxJ class `TCPType`). Note that unlike UDP, TCP uses the same format for transmitting and receiving streams. Fields requiring manual handling by the developer are further explained below. An overview of the automatically handled fields common to all MaxCompiler network streams is provided in *subsection 3.4*.

- The **socket** field contains a socket number ranging from 0 to 63 that identifies the TCP connection associated with the data, which is useful if an application needs to communicate concurrently with

Field	Offset (bits)	Width (bits)
data	0	64
socket	64	8
mod	72	3
eof	75	1
sof	76	1

Table 8: TCP stream format

multiple peers and multiplex the traffic from all of them over a single stream. Applications can use a CPU API function (*subsubsection 6.3.5*) to obtain the socket number for any open connection. Although the socket number may vary between segments, it is always the same throughout any given segment. For transmitting streams the socket number therefore only needs to be set at the start of the segment, when SOF is true. For receiving streams, the socket number is valid during the whole segment.

6.2 Field Accumulator Adaptions for TCP

The original field accumulator example as introduced in *subsection 3.2* uses UDP as an underlying protocol. Only very few modifications to the Kernel and Manager are necessary to switch to TCP.

In the Kernel we change the framed link type of the input and output to TCP as follows:

```
FrameData<DataIn> frameIn = io.frameInput("frameIn", new DataIn(), new TCPType());
```

```
FrameData<DataOut> frameOut = new FrameData<DataOut>(this, new DataOut(), new TCPType());

frameOut["sum"] <== sum;
frameOut["totalItems"] <== totalItems;
frameOut. linkfield [TCPType.SOCKET] <==
    frameIn. linkfield [TCPType.SOCKET];
```

The new Manager configuration simply connects the Kernel IO to the TCP stream on SFP1:

```
m.setIO(link("frameIn",   TCP(NetworkConnection.CH2_SFP1)),
        link ("frameOut", TCP(NetworkConnection.CH2_SFP1)));
```

The TCP stream can now be configured from the CPU as explained in the following sections. *Listing 9* and *Listing 10* show the TCP field accumulator CPU code. Note that general structure still closely resembles the original UDP version, although different SLiC API calls are used for the TCP configuration.

Listing 9: TCP Field accumulator example CPU code (FieldAccumulatorCpuCode.c)

```
#define _GNU_SOURCE

#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>

#include <MaxSLiCInterface.h>
#include "FieldAccumulatorTCP.h"


typedef struct {
    uint32_t total_items;
    uint32_t sum;
} __attribute__ ((__packed__)) frame_t;


static int create_cpu_tcp_socket(struct in_addr *remote_ip, int port) {
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    int state = 1;
    setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &state, sizeof(state));

    struct sockaddr_in cpu;
    memset(&cpu, 0, sizeof(cpu));
    cpu.sin_family = AF_INET;
    cpu.sin_port = htons(port);
    cpu.sin_addr = *remote_ip;

    connect(sock, (const struct sockaddr*) &cpu, sizeof(cpu));

    return sock;
}


static void exchangeItems(int sock, const uint8_t* items, uint8_t num_items) {
    static uint32_t total_items = 0;

    uint8_t data_to_send[1 + UINT8_MAX];
    data_to_send[0] = num_items;
    memcpy(data_to_send + 1, items, num_items);

    send(sock, &data_to_send, num_items + 1, 0);
    frame_t data_received;

    recv(sock, &data_received, sizeof(data_received), 0);

    printf("Received: total_items = %u, sum = %u\n", data_received.total_items, data_received.sum);

    total_items += num_items;

    uint32_t sum = 0;
    for (int i = 0; i < num_items; i++)
        sum += items[i];

    if (data_received.total_items != total_items || data_received.sum != sum) {
        printf("Error! Expected: total_items = %u, sum = %u\n", total_items, sum);
        exit(1);
    }
}
```

Listing 10: TCP Field accumulator example CPU code (FieldAccumulatorCpuCode.c)

```
int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf ("Usage: %s <dfe_ip> <cpu_ip> <netmask>\n", argv[0]);
        return 1;
    }

    struct in_addr dfe_ip;
    inet_aton(argv[1], &dfe_ip);
    struct in_addr cpu_ip;
    inet_aton(argv[2], &cpu_ip);
    struct in_addr netmask;
    inet_aton(argv[3], &netmask);
    const int port = 5007;

    max_file_t *maxfile = FieldAccumulatorTCP_init();
    max_engine_t *engine = max_load(maxfile, "*");

    max_ip_config(engine, MAX_NET_CONNECTION_CH2_SFP1, &dfe_ip, &netmask);

    max_tcp_socket_t *dfe_socket = max_tcp_create_socket(engine, "tcp_ch2_sfp1");
    max_tcp_listen(dfe_socket, port);
    max_tcp_await_state(dfe_socket, MAX_TCP_STATE_LISTEN, NULL);

    int cpu_socket = create_cpu_tcp_socket(&dfe_ip, port);

    uint8_t items1[] = { 2, 5, 8, 1 };
    exchangeItems(cpu_socket, items1, sizeof(items1));

    uint8_t items2[] = { 50, 2 };
    exchangeItems(cpu_socket, items2, sizeof(items2));

    uint8_t items3[] = { 1, 2, 3, 4, 5, 6 };
    exchangeItems(cpu_socket, items3, sizeof(items3));

    close(cpu_socket);

    max_tcp_close(dfe_socket);

    max_unload(engine);
    max_file_free(maxfile);

    return 0;
}
```

6.3 CPU API for TCP

An overview of functions generally sufficient to deploy an application using TCP successfully is provided in this section for reference. Some are illustrated by the field accumulator example. Although the API is designed for similarity with standard socket programming conventions for the most part, there are some notable differences.

- Functions are non-blocking unless otherwise stated.
- All sockets needed during the entire run of the application must be created before any of them is used.

Details are given below.

6.3.1 Creating a Socket

The current state and configuration of the TCP processing hardware are tracked in software by a data structure known as a **socket** and referenced by a pointer of type `max_tcp_socket_t`. The following blocking function performs some initialization on the hardware the first time it is called, and returns a reference to a socket.

max_tcp_socket_t max_tcp_create_socket (max_engine_t ∗engine, **const char** ∗stream_name)*

The parameters have these interpretations.

- `engine` identifies the DFE to use, and is given by a prior call to `max_load()`.
- `stream_name` is a name of the relevant TCP stream specified in the Manager code.

In the field accumulator example, a socket is created for the TCP stream on network connection SFP1, identified by the automatically generated stream name `"tcp_ch2_sfp1"`:

```
max_tcp_socket_t ∗dfe_socket = max_tcp_create_socket(engine, "tcp_ch2_sfp1");
```

6.3.2 Starting an Outward Connection

Starting a TCP connection is a prerequisite for transferring data on it. The following function starts a TCP connection, such as a client might connect to a server, using a previously created socket given by the `tcp` parameter. The other parameters have the usual interpretations, namely the IP address and the port number on the remote peer.

***void** max_tcp_connect (max_tcp_socket_t ∗tcp, **const struct** in_addr ∗remote_ip, uint16_t remote_port)*

After this connection is made, Kernels and state machines can send to it by including the associated socket number in outgoing TCP streams as shown in *Table 8*. This socket number can be ascertained by `max_tcp_get_socket_number` as explained in *subsubsection 6.3.5* and passed from the CPU to a Kernel as a scalar input.

Another alternative that avoids the scalar input is to use a hard coded socket number in both the CPU code and the Kernel. In this case the following blocking function is appropriate.

max_tcp_socket_t max_tcp_create_socket_with_number (max_engine_t ∗engine, **const char** ∗stream_name, uint16_t socket_number)*

The `socket_number` parameter should be a number in the range 0 to 63.

6.3.3 Accepting an Inward Connection

The `max_tcp_listen` function allows a connection to be made by a remote peer, typically as a server would accept a connection from a client.

void *max_tcp_listen (max_tcp_socket_t ∗socket, uint16_t local_port)*

The `socket` parameter is an existing socket for the dataflow engine accepting the connection. The `local_port` parameter is the port number the remote peer must specify to establish the connection and ensure that TCP traffic is routed to it.

Applications that accept multiple connections on the same port should call this function once for each possible connection (up to a maximum of 64) with a different socket parameter for each call.

✘ This method of enabling multiple incoming connections to the same port is specific to MaxCompiler and differs considerably from standard BSD socket programming conventions.

The field accumulator example accepts inward connections on the previously created socket and uses a user-specified local port:

```
85	max_tcp_listen(dfe_socket, port);
```

Note that because `max_tcp_listen` call is non-blocking, the socket may not be in the listening state yet when the function returns. *subsubsection 6.3.6* shows how an application can wait for certain TCP state changes to occur.

6.3.4 Closing a Connection

The following functions close a connection previously opened by `max_tcp_connect` (for example, as a client closing a connection to a server) or previously accepted by `max_tcp_listen`. Closing a connection disables the exchange of further data through it, although the socket may be reused for other connections.

void *max_tcp_close(max_tcp_socket_t ∗socket)*

void *max_tcp_close_advanced (max_tcp_socket_t ∗socket, max_tcp_close_mode_t close_mode)*

The `socket` parameter identifies the socket to be closed. The `close_mode` is an enumerated type with these possible values and interpretations.

- `MAX_TCP_CLOSE_GRACEFUL` - Close the connection with standard handshaking acknowledgments in both directions (default).
- `MAX_TCP_CLOSE_ABORT_RESET` - Close the connection in a way that informs the remote peer of an anomalous condition and requests no further acknowledgment.
- `MAX_TCP_CLOSE_ABORT_NO_RESET` - Close the connection abruptly without any advice to the peer.

Aborting a connection by either of the last two modes may cause some data to be lost. Any data received from the network but not yet processed by the DFE is lost when a connection is aborted.

The field accumulator example closes its socket using the basic `max_tcp_close` function, after all data has been processed but before closing the dataflow engine:

```
max_tcp_close(dfe_socket);
```

6.3.5 Interrogating a Connection

The blocking `max_tcp_status` function can be used to ascertain the status of a connection given the socket.

void *max_tcp_status (max_tcp_socket_t ∗socket, max_tcp_connection_status_t ∗status)*

The `max_tcp_status` function modifies a `max_tcp_connection_status_t` structure of this form.

```
typedef struct {
    max_tcp_connection_state_t state;
    uint32_t  input_sequence;
    uint32_t  output_sequence;
    uint32_t  window_space_left;
    struct in_addr  remote_ip;
    uint16_t  remote_port;
    struct ether_addr remote_mac;
} max_tcp_connection_status_t
```

The state is an enumerated type with one of the values explained below. The names follow standard TCP state name conventions, except for `MAX_TCP_STATE_CLOSED_DATA_PENDING`, which is specific to MaxCompiler.

- When a socket is first created by `max_tcp_create`, it is in `MAX_TCP_STATE_CLOSED`. It returns to this state after the application calls `max_tcp_close` on it, or when the remote peer breaks the connection.
- When a closed connection is opened for listening, it enters `MAX_TCP_STATE_LISTEN`, and remains that way unless a remote peer connects to the port.
- A connection enters `MAX_TCP_STATE_ESTABLISHED` when a remote peer connects to a listening port, or when a connection started by the local system with `max_tcp_connect` is accepted by a remote peer.
- `MAX_TCP_STATE_CLOSE_WAIT` is a normally transient state that occurs between the time the local application closes a connection and the remote peer acknowledges that the connection is closed. A connection could persist in this state if the remote peer becomes unreachable.
- `MAX_TCP_STATE_CLOSED_DATA_PENDING` is a state in which a connection has been closed but the DFE contains some data not yet read by the application. A correct application should always read the remaining data in this case.

One example of how to use this information is in a server with many clients connecting intermittently. Each time a connection is closed by a client, the server detects the closure and changes the connection back to the `LISTEN` state by calling `max_tcp_listen`.

Another item of useful information that can be queried about a socket is its socket number. This number is embedded as the connection field in each word transmitted via the socket through a stream, as shown in *Table 8*, so that Kernels can distinguish among multiple connections on the same stream. The socket number for any previously created socket is returned by the following blocking function.

uint16_t max_tcp_get_socket_number(max_tcp_socket_t ∗socket)

6.3.6 Monitoring Connections

The `max_tcp_select` function can detect a change in the state of a connection so that the application can take appropriate action. When called, it blocks until a change occurs on any of a specified set of sockets or a timeout is reached. When returning, it reports the changed sockets in an array. If any changes to sockets in the set occur between calls or before the first call, they are reported immediately as of the next call. This function also modifies the timeout parameter to reflect the time remaining if it returns before the time limit expires.

```
void max_tcp_select(
    uint16_t num_sockets,
    max_tcp_socket_t *sockets[],
    uint64_t *num_changed_sockets,
    max_tcp_socket_t *changed_sockets[],
    struct timeval *timeout);
```

The parameters have these interpretations.

- `num_sockets` is the number of sockets in the `sockets` array.
- The `sockets` parameter is an array of sockets whose status is to be monitored, initialized by the caller.
- `timeout` is a pointer to a standard `timeval` structure as defined in `sys/time.h` indicating the length of time to wait for a change in status, or one of these special values:
  - If `timeout` is zero, `max_tcp_select` returns immediately, in effect polling the dataflow engine.
  - If `timeout` is a `NULL` pointer, `max_tcp_select` waits indefinitely.

  If the state changes before the time limit is reached, `*timeout` is modified to store the time remaining.
- `changed_sockets` is the address of an array wherein the function returns the sockets whose states have changed. An array of sufficient size should be allocated in advance by the caller.
- `num_changed_sockets` is the number of sockets in the array referenced by `changed_sockets`.

The `max_tcp_select` function is meant to be used in combination with `max_tcp_status` (*subsubsection 6.3.5*), because state changes are uninformative in themselves unless the current state is known. A typical application might execute a main loop wherein it repeatedly suspends itself by calling `max_tcp_select`, and calls `max_tcp_status` whenever it wakes up, so as to determine the nature of the state change and act on it.

Another way of monitoring a socket is by the following function, which blocks until a socket reaches a specified state or a timeout is reached.

```
int max_tcp_await_state (max_tcp_socket_t *socket, max_tcp_connection_state_t state, struct timeval *timeout)
```

The `max_tcp_await_state` function monitors only a single socket given by the `socket` parameter.

- If the `timeout` parameter is `NULL`, the state of the socket is necessarily that of the `state` parameter when `max_tcp_await_state` returns.

- If the `timeout` parameter is not `NULL`, the state might not be reached.
  - The function returns zero if the state is reached, and non-zero if the state is not reached.
  - The `timeout` parameter is modified to reflect the time remaining.

The field accumulator example waits without a timeout for its socket to reach the listening state using the following line:

```
    max_tcp_await_state(dfe_socket, MAX_TCP_STATE_LISTEN, NULL);
```

7

Network Managers

Examples in previous chapters have used the Standard Manager API, whose simplicity is suited for rapid deployment of basic network applications. However, advanced users may prefer the Network Manager API as an alternative to the Standard Manager for more specific control over some aspects of the design.

The main difference between the Standard Manager API and the Network Manager API is that the Network Manager allows more general patterns of interconnection among multiple Manager blocks. Blocks in a Network Manager are connected by named streams to inputs or outputs on other blocks explicitly chosen by the designer.

Network Managers are an extension of the Custom Managers used in non-networking DFE designs and share many of their features. For a general introduction to Custom Managers please refer to the MaxCompiler Manager tutorial. The CPU API for networking DFE designs remains the same no matter whether a design was created using a Network Manager or a Standard Manager.

An example application illustrating the Network Manager networking features is shown at the end of this chapter in *subsection 7.2*.

7.1 Network Streams

Manager blocks such as Kernels and State Machines are connected to the network through protocol-specific bi-directional streams. The attributes common to all streams are a string name used to identify the stream in the CPU code and the physical network connection (SFP1 or SFP2) through which the stream communicates to remote peers.

MaxCompiler currently supports the network stream classes `UDPStream`, `TCPStream` and `EthernetStream`, which are explained in the following sections along with any protocol-specific configuration options.

As all network streams are bi-directional, they provide methods to retrieve their receiving and transmitting ends (`getReceiveStream()` and `getTransmitStream()`), which can be connected directly to Manager block inputs and outputs, respectively. This is shown in the following code using an UDP stream as an example:

```
UDPStream udp_stream = ...;
DFELink receive = udp_stream.getReceiveStream();
DFELink transmit = udp_stream.getTransmitStream();
```

Note that it is possible to leave either the receiving or the transmitting end unconnected but not both.

7.1.1 UDP Streams

A UDP stream is added to a Network Manager using the following call:

```
UDPStream addUDPStream(
    String name,
    NetworkConnection connection,
    UDPConnectionMode connection_mode,
    UDPChecksumMode receive_checksum_mode);
```

The `name` and `connection` parameters are interpreted according to *subsection 7.1*. `connection_mode` specifies whether to create a one-to-one or one-to-many UDP stream (see *section 5*) and `receive_checksum_mode` decides how to handle incoming packets with an incorrect UDP checksum.

Packets with an incorrect UDP checksum can be dropped automatically on receipt by the MaxCompiler networking infrastructure (`DropBadFrames`) or flagged as incorrect (`FlagOnEOF`) using the checksum_bad stream field (see *subsubsection 5.1.2*). Checksum processing can also be disabled completely by supplying `Disabled` as an argument.

7.1.2 TCP Streams

To create a TCP stream, it is sufficient to specify a stream name and network connection as explained in *subsection 7.1*:

```
TCPStream addTCPStream(
    String name,
    NetworkConnection connection);
```

Further TCP options are available through the `NetwokManager.network_config` object. The TCP window sized can be controlled with the following methods of `NetworkManager.network_config`:

```
void setTCPReceiveWindowSize(
    NetworkConnection connection,
    int   size_in_kilobytes ) ;

void setTCPTransmitWindowSize(
    NetworkConnection connection,
    int   size_in_kilobytes ) ;
```

Note that the window sizes are specified per network connection and not per TCP stream.

7.1.3 Ethernet Streams

The following method creates an Ethernet stream:

EthernetStream addEthernetStream(
  ***String** name,*
  NetworkConnection connection,
  EthernetChecksumMode receive_checksum_mode);

The `name` and `connection` parameters have the same interpretation as above. `receive_checksum_mode` is equivalent to the corresponding parameter to UDP streams (*subsubsection 7.1.1*), with the only difference that checksum verification cannot be disabled for Ethernet streams.

7.2 Key-Value Example

We now introduce a new example to illustrate the Network Manager features explained above. The key-value application consists of two Kernels, one after the other. The preprocessor Kernel receives frames through a UDP stream on SFP1, each packet containing key and a value field. It forwards the value of packets whose key matches certain criteria to the adder Kernel. For each value the adder Kernel receives it reads another value from sent from the CPU code via PCIe, adds the two values and sends the result over a TCP stream on SFP1. *Figure 14* shows the design of the key-value example.

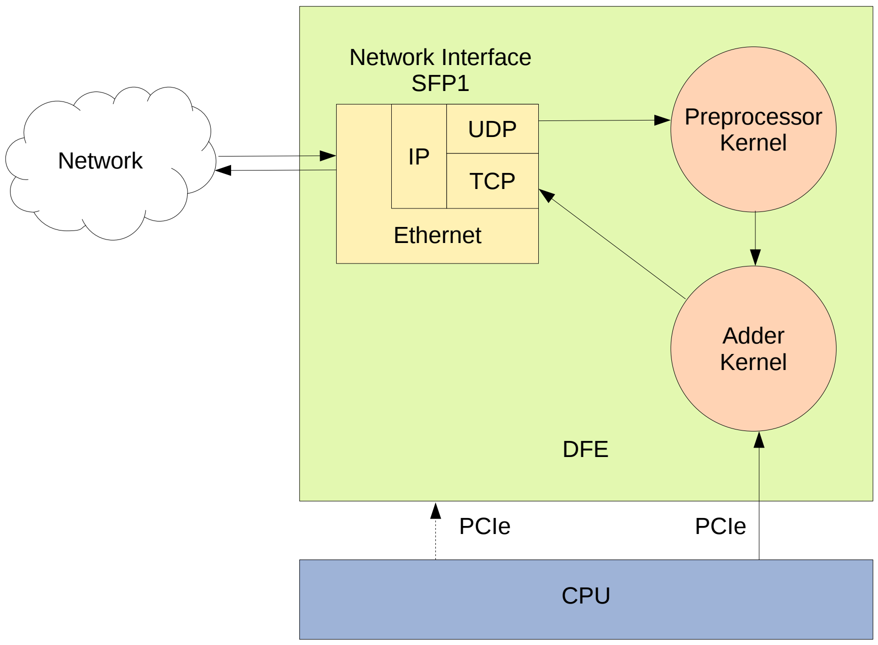


Figure 14: Block diagram of the key-value example

Similar to previous examples, the CPU code for the key-value application sets up the DFE, establishes the connections for the network streams and exchanges data with the DFE via BSD sockets. Additional data for the adder Kernel is transferred to the DFE via a PCIe stream.

7.2.1 Kernels

The following paragraphs provide a summary of the two Kernels used in the key-value example. For a full introduction to Framed Kernels please refer to *section 3*.

Preprocessor Kernel The preprocessor Kernel defines different frame formats for incoming and outgoing frames. Incoming frames consist of a key and an integer value field, outgoing frames only carry a floating point value:

```
private static class KeyValFormat extends FrameFormat {
    public KeyValFormat() {
        super(ByteOrder.LITTLE_ENDIAN);
        addField("key", dfeUInt(8));
        addField("value", dfeInt(32));
    }
}

public static class ValFormat extends FrameFormat {
    public ValFormat() {
        super(ByteOrder.LITTLE_ENDIAN);
        addField("value", dfeFloat(8, 24));
    }
}
```

An input and a FrameData instance for the output is instantiated using the two frame formats. The value field of the incoming frame is cast to a floating point number and assigned to the value field of the outgoing frame. As the output of the preprocessor Kernel is passed on to another Kernel rather than to a network interface, we use the `SimpleFramedLinkType` Link format when instantiating the output:

```
    FrameData<KeyValFormat> keyValIn =
        io.frameInput("keyValIn", new KeyValFormat(), new UDPOneToOneRXType());

    FrameData<ValFormat> valOut =
        new FrameData<ValFormat>(this, new ValFormat(), new SimpleFramedLinkType());

    valOut["value"] <== keyValIn["value"].cast(valOut["value"].getType());
```

The Kernel's output is only enabled if the key is equal to the character "X":

```
    DFEVar enable = keyValIn["key"] === 'X';

    io.frameOutput(
        "valOut",
        valOut,
        enable,
        keyValIn.isStart());
```

The full source code for the preprocessor Kernel is shown in *Listing 11*

Adder Kernel The adder Kernel reuses the output frame format of the preprocessor kernel to instantiate an input and a FrameData instance for the output:

```
    ValFormat valFormat = new ValFormat();
    FrameData<ValFormat> valIn =
        io.frameInput("valIn", valFormat, new SimpleFramedLinkType());

    FrameData<ValFormat> valOut =
        new FrameData<ValFormat>(this, valFormat, new TCPType());
```

Additionally, a regular Kernel input is created to receive values directly from the CPU:

```
    io.pushInputRegistering(false);
    DFEVar cpu_value = io.input("cpuValIn", dfeFloat(8, 24), valIn.isStart());
```

Note that input registering must be disabled when mixing framed and regular inputs.

Output frames are generated by setting their value field to the sum of the values received through

Listing 11: Preprocessor Kernel (PreprocessorKernel.maxj)

```
package keyval;

import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.networking.v1.framed_kernels.ByteOrder;
import com.maxeler.networking.v1.framed_kernels.FrameData;
import com.maxeler.networking.v1.framed_kernels.FrameFormat;
import com.maxeler.networking.v1.framed_kernels.FramedKernel;
import com.maxeler.networking.v1.kernel_types.SimpleFramedLinkType;
import com.maxeler.networking.v1.kernel_types.UDPOneToOneRXType;

public class PreprocessorKernel extends FramedKernel {

    private static class KeyValFormat extends FrameFormat {
        public KeyValFormat() {
            super(ByteOrder.LITTLE_ENDIAN);
            addField("key", dfeUInt(8));
            addField("value", dfeInt(32));
        }
    }

    public static class ValFormat extends FrameFormat {
        public ValFormat() {
            super(ByteOrder.LITTLE_ENDIAN);
            addField("value", dfeFloat(8, 24));
        }
    }

    PreprocessorKernel(KernelParameters parameters) {
        super(parameters);

        FrameData<KeyValFormat> keyValIn =
            io.frameInput("keyValIn", new KeyValFormat(), new UDPOneToOneRXType());

        FrameData<ValFormat> valOut =
            new FrameData<ValFormat>(this, new ValFormat(), new SimpleFramedLinkType());

        valOut["value"] <== keyValIn["value"].cast(valOut["value"].getType());

        DFEVar enable = keyValIn["key"] === 'X';

        io.frameOutput(
            "valOut",
            valOut,
            enable,
            keyValIn.isStart());
    }
}
```

the framed input and the regular input. The TCP socket for the outgoing frame is assumed to be always 0 (this is set up by the CPU code explained below):

```
        valOut["value"] <== valIn["value"] + cpu_value;
        valOut.linkfield[TCPType.SOCKET] <==
            constant.var(valOut.linkfield[TCPType.SOCKET].getType(), 0);

        io.frameOutput("valOut", valOut);
```

See *Listing 12* for the full source code of the adder Kernel.

Listing 12: Adder Kernel (AdderKernel.maxj)

```
package keyval;

import keyval.PreprocessorKernel.ValFormat;

import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
import com.maxeler.networking.v1.framed_kernels.FrameData;
import com.maxeler.networking.v1.framed_kernels.FramedKernel;
import com.maxeler.networking.v1.kernel_types.SimpleFramedLinkType;
import com.maxeler.networking.v1.kernel_types.TCPType;


public class AdderKernel extends FramedKernel {

    AdderKernel(KernelParameters parameters) {
        super(parameters);

        ValFormat valFormat = new ValFormat();
        FrameData<ValFormat> valIn =
            io.frameInput("valIn", valFormat, new SimpleFramedLinkType());

        FrameData<ValFormat> valOut =
            new FrameData<ValFormat>(this, valFormat, new TCPType());

        io.pushInputRegistering(false);
        DFEVar cpu_value = io.input("cpuValIn", dfeFloat(8, 24), valIn.isStart());

        valOut["value"] <== valIn["value"] + cpu_value;
        valOut.linkfield[TCPType.SOCKET] <==
            constant.var(valOut.linkfield[TCPType.SOCKET].getType(), 0);

        io.frameOutput("valOut", valOut);
    }
}
```

7.2.2 Manager

The Manager instantiates Kernel blocks for the preprocessor and adder kernels explained above:

```
15        KernelBlock preprocessor = addKernel(new PreprocessorKernel(makeKernelParameters("Preprocessor")));
16        KernelBlock adder = addKernel(new AdderKernel(makeKernelParameters("Adder")));
```

The output of the preprocessor Kernel is then connected to the “valIn” output of the adder Kernel:

```
18        adder.getInput("valIn") <== preprocessor.getOutput("valOut");
```

A newly created stream from the CPU is connected to the adder Kernel’s remaining “cpuValue” input:

```
20        adder.getInput("cpuValIn") <== addStreamFromCPU("cpuValue");
```

Finally we create a one-to-one UDP stream and a TCP stream on SFP1. The received UDP packets

Listing 13: Key-Value Manager (KeyValManager.maxj)

```
package keyval;

import com.maxeler.maxcompiler.v2.managers.custom.blocks.KernelBlock;
import com.maxeler.networking.v1.managers.NetworkManager;
import com.maxeler.networking.v1.managers.netlib.Max3NetworkConnection;
import com.maxeler.networking.v1.managers.netlib.TCPStream;
import com.maxeler.networking.v1.managers.netlib.UDPChecksumMode;
import com.maxeler.networking.v1.managers.netlib.UDPConnectionMode;
import com.maxeler.networking.v1.managers.netlib.UDPStream;

public class KeyValManager extends NetworkManager {
    public KeyValManager(String[] args) {
        super(new KeyValEngineParameters(args));

        KernelBlock preprocessor = addKernel(new PreprocessorKernel(makeKernelParameters(”Preprocessor”)));
        KernelBlock adder = addKernel(new AdderKernel(makeKernelParameters(”Adder”)));

        adder.getInput(”valIn”) <== preprocessor.getOutput(”valOut”);

        adder.getInput(”cpuValIn”) <== addStreamFromCPU(”cpuValue”);

        UDPStream udpStream = addUDPStream(
            ”udp_ch2_sfp1”,
            Max3NetworkConnection.CH2_SFP1,
            UDPConnectionMode.OneToOne,
            UDPChecksumMode.DropBadFrames);

        preprocessor.getInput(”keyValIn”) <== udpStream.getReceiveStream();

        TCPStream tcpStream = addTCPStream(
                ”tcp_ch2_sfp1”,
                Max3NetworkConnection.CH2_SFP1);

        tcpStream.getTransmitStream() <== adder.getOutput(”valOut”);
    }

    public static void main(String[] args) {
        KeyValManager m = new KeyValManager(args);
        m.build();
    }
}
```

are directed to the preprocessor Kernel and frames produces by adder Kernel are sent over TCP:

```
        UDPStream udpStream = addUDPStream(
            ”udp_ch2_sfp1”,
            Max3NetworkConnection.CH2_SFP1,
            UDPConnectionMode.OneToOne,
            UDPChecksumMode.DropBadFrames);

        preprocessor.getInput(”keyValIn”) <== udpStream.getReceiveStream();

        TCPStream tcpStream = addTCPStream(
                ”tcp_ch2_sfp1”,
                Max3NetworkConnection.CH2_SFP1);

        tcpStream.getTransmitStream() <== adder.getOutput(”valOut”);
```

Listing 13 shows the full Manager source code.

7.2.3 CPU Application

The CPU application performs the basic DFE configuration as explained in *section 4*. It then creates a UDP and a listening TCP socket on the DFE and waits for the TCP socket to reach the listening state:

```
max_ip_config(engine, MAX_NET_CONNECTION_CH2_SFP1, &dfe_ip, &netmask);
max_udp_socket_t *dfe_udp_socket = max_udp_create_socket(engine, "udp_ch2_sfp1");
max_udp_bind(dfe_udp_socket, udp_port);
max_udp_connect(dfe_udp_socket, &cpu_ip, udp_port);

max_tcp_socket_t *dfe_tcp_socket = max_tcp_create_socket_with_number(engine, "tcp_ch2_sfp1", 0);
max_tcp_listen(dfe_tcp_socket, tcp_port);
max_tcp_await_state(dfe_tcp_socket, MAX_TCP_STATE_LISTEN, NULL);
```

Note that the TCP socket number is explicitly set to 0, according to the assumption made in the adder Kernel.

Once the TCP socket has reached the listening state the CPU code establishes a UDP and TCP connection to the DFE using BSD sockets and configures a PCIe stream. Data is then sent to the DFE via the UDP connection and the PCIe stream and received through the TCP connection. This is shown in full CPU source code in *Listing 14*, *Listing 15* and *Listing 16*.

After all data has been transferred, the DFE sockets are closed:

```
max_udp_close(dfe_udp_socket);
max_tcp_close(dfe_tcp_socket);
```

Listing 14: Key-Value CPU code (KeyValCpuCode.c)

```
#define _GNU_SOURCE

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <linux/if.h>
#include <linux/if_ether.h>
#include <netinet/ether.h>
#include <netinet/tcp.h>
#include <sys/ioctl.h>
#include <stdbool.h>

#include <MaxSLiCInterface.h>
#include "KeyVal.h"


typedef struct key_val_s {
     uint8_t  key;
     int32_t  value;
} __attribute__  ((__packed__))  key_val_t;

static int create_cpu_udp_socket(struct in_addr *local_ip, struct in_addr *remote_ip, int port) {
     int sock = socket(AF_INET, SOCK_DGRAM, 0);

     struct sockaddr_in cpu;
     memset(&cpu, 0, sizeof(cpu));
     cpu.sin_family = AF_INET;
     cpu.sin_port = htons(port);

     cpu.sin_addr = *local_ip;
     bind(sock, (struct sockaddr *)&cpu, sizeof(cpu));

     cpu.sin_addr = *remote_ip;
     connect(sock, (const struct sockaddr*) &cpu, sizeof(cpu));

     return sock;
}

static int create_cpu_tcp_socket(struct in_addr *remote_ip, int port) {
     int sock = socket(AF_INET, SOCK_STREAM, 0);

     int state = 1;
     setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &state, sizeof(state));

     struct sockaddr_in cpu;
     memset(&cpu, 0, sizeof(cpu));
     cpu.sin_family = AF_INET;
     cpu.sin_port = htons(port);
     cpu.sin_addr = *remote_ip;

     connect(sock, (const struct sockaddr*) &cpu, sizeof(cpu));

     return sock;
}
```

Listing 15: Key-Value CPU code (KeyValCpuCode.c)

```
void exchangeFrames(max_file_t *maxfile, max_engine_t *engine, int udp_sock, int tcp_sock) {
    const int num_frames = 64;
    key_val_t key_val_in;
    float val_out;

    float *val_in = malloc(sizeof(*val_in) * (num_frames / 2));
    for (int j = 0; j < num_frames / 2; j++)
        val_in[j] = j * 10;

    max_actions_t *actions = max_actions_init(maxfile, NULL);
    max_queue_input(actions, "cpuValue", val_in, sizeof(*val_in) * (num_frames/2));
    max_run_t *run = max_run_nonblock(engine, actions);

    for (int j = 0; j < num_frames; j++) {
        key_val_in.key = j % 2 == 0 ? 'X' : 'Y';
        key_val_in.value = j;

        send(udp_sock, &key_val_in, sizeof(key_val_t), 0);
        printf("Sent: key = %c value = %d ", key_val_in.key, key_val_in.value);

        if (key_val_in.key != 'X') {
            printf("\n");
            continue;
        }

        recv(tcp_sock, &val_out, sizeof(float), 0);
        printf("Received: value = %f\n", val_out);

        if (val_out != key_val_in.value + val_in[j / 2]) {
            printf("Error!\n");
            exit(1);
        }
    }

    max_wait(run);
    max_actions_free(actions);
    free(val_in);
}
```

Listing 16: Key-Value CPU code (KeyValCpuCode.c)

```
int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf ("Usage: %s <dfe_ip> <cpu_ip> <netmask>\n", argv[0]);
        return 1;
    }

    struct in_addr dfe_ip;
    inet_aton(argv[1], &dfe_ip);
    struct in_addr cpu_ip;
    inet_aton(argv[2], &cpu_ip);
    struct in_addr netmask;
    inet_aton(argv[3], &netmask);
    const int udp_port = 5007;
    const int tcp_port = 5008;

    max_file_t *maxfile = KeyVal_init();
    max_engine_t * engine = max_load(maxfile, "*");

    max_ip_config(engine, MAX_NET_CONNECTION_CH2_SFP1, &dfe_ip, &netmask);
    max_udp_socket_t *dfe_udp_socket = max_udp_create_socket(engine, "udp_ch2_sfp1");
    max_udp_bind(dfe_udp_socket, udp_port);
    max_udp_connect(dfe_udp_socket, &cpu_ip, udp_port);

    max_tcp_socket_t *dfe_tcp_socket = max_tcp_create_socket_with_number(engine, "tcp_ch2_sfp1", 0);
    max_tcp_listen(dfe_tcp_socket, tcp_port);
    max_tcp_await_state(dfe_tcp_socket, MAX_TCP_STATE_LISTEN, NULL);

    int cpu_udp_socket = create_cpu_udp_socket(&cpu_ip, &dfe_ip, udp_port);
    int cpu_tcp_socket = create_cpu_tcp_socket(&dfe_ip, tcp_port);

    exchangeFrames(maxfile, engine, cpu_udp_socket, cpu_tcp_socket);

    max_udp_close(dfe_udp_socket);
    max_tcp_close(dfe_tcp_socket);

    max_unload(engine);
    max_file_free(maxfile);

    return 0;
}
```

8

Ethernet Frame Processing

This chapter summarizes MaxCompiler features pertaining to raw Ethernet frame processing. Ethernet frame processing is normally implicit in higher level protocols such as TCP or UDP, but raw Ethernet handling might be of interest For low level network traffic analysis applications, or where it is desirable to implement other higher layer protocols. Support for Ethernet frame processing by MaxCompiler includes stream format definitions and compiler generated Ethernet protocol handling hardware. These features are documented in this section and demonstrated in a time stamping application.

8.1 Receiving Streams

A stream for carrying Ethernet frames received from the network has the format shown in *Table 9* (MaxJ class `EthernetRXType`). Fields requiring manual handling by the developer are further explained below. An overview of the automatically handled fields common to all MaxCompiler network streams is provided in *subsection 3.4*.

Note however that when using Ethernet streams, the data field is used to carry both the Ethernet header and payload. The standard Ethernet header fields (destination MAC, source MAC and Ethertype) must be handled manually by the developer. All other parts of the Ethernet frame, including the Frame Check Sequence, are handled implicitly by MaxCompiler and therefore not accessible.

- **checksum_bad** is true only if the current frame is found to contain an error based on the checksum, and manual checksum verification is enabled. If automatic checksum verification is used

Field	Offset (bits)	Width (bits)
data	0	64
eof	64	1
sof	65	1
mod	66	3
checksum_bad	69	1

Table 9: Ethernet receive stream format

Field	Offset (bits)	Width (bits)
data	0	64
eof	64	1
sof	65	1
mod	66	3

Table 10: Ethernet transmit stream format

(the default), this field is always false as bad frames are dropped before reaching the Kernel. The checksum_bad field is defined whenever EOF is true.

8.2 Transmitting Streams

The format for streams carrying Ethernet traffic outward to the network is depicted in *Table 10* (MaxJ class `EthernetTXType`). This format is similar to that of receiving streams but lacks a checksum bit. The remaining metadata fields have similar interpretations, and the application from which the stream originates is responsible for ensuring that they are defined consistently with the specification described in *subsection 8.1*.

8.3 Time Stamped Packet Capture Example

The network diagnostic tool depicted in *Figure 15* captures Ethernet packets from the network, notes the arrival time of each one, and logs them in a file for further analysis.

This application exemplifies a simple case of Ethernet frame processing. The time stamp block receives Ethernet frames from the Ethernet block by way of a stream using the format described in *subsection 8.1*. It increments a counter on every cycle, and attaches that count to each word passed to the CPU through another stream connected to the PCIe bus. The time stamp block is written by the designer, and the Ethernet block is a standard component generated automatically by MaxCompiler.

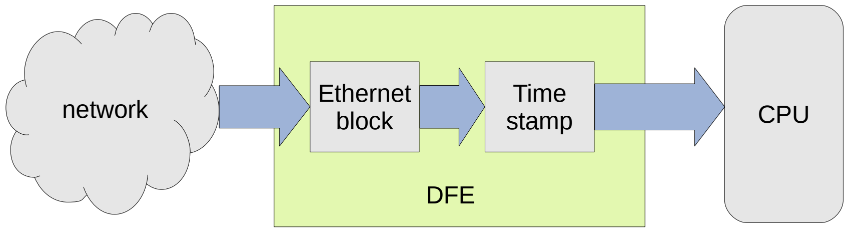


Figure 15: Time stamping application

Listing 17: State Machine code for the time stamp example

```
package stamp;

import com.maxeler.maxcompiler.v2.managers.DFEManager;
import com.maxeler.maxcompiler.v2.statemachine.DFEsmStateValue;
import com.maxeler.maxcompiler.v2.statemachine.manager.ManagerStateMachine;
import com.maxeler.maxcompiler.v2.statemachine.manager.DFEsmPushInput;
import com.maxeler.maxcompiler.v2.statemachine.manager.DFEsmPushOutput;

public class StampStateMachine extends ManagerStateMachine {

    private final DFEsmStateValue counter = state.value(dfeUInt(48), 0);

    private final DFEsmPushInput input = io.pushInput("eth_to_sm", dfeUInt(70), 1);
    private final DFEsmPushOutput output = io.pushOutput("stamped_eth_frm_sm", dfeUInt(128), 1);

    public StampStateMachine(DFEManager owner) {
        super(owner);
    }

    @Override
    protected void nextState() {
        counter.next <== counter + 1;
    }

    @Override
    protected void outputFunction() {
        output <== constant.value(dfeUInt(10), 0) # counter # input;
        output.valid <== input.valid;
        input.stall <== output.stall;
    }
}
```

Time stamping state machine In order to update the count continually without ever stalling, the time stamp block shown *Figure 15* is implemented as a Manager state machine rather than a Kernel. State machine programming is explained fully in the **MaxCompiler State Machine Tutorial** document, but this example requires only a basic understanding of state machines.

The source code is shown in *Listing 17* for the time stamping state machine. The state consists only of the `counter` variable. The control signals are passed through in both directions. The output is made by concatenating the input with the count and some padding using the # operator.

Manager The Network Manager interface is used to instantiate the time stamping state machine as shown in the following line:

```
        StateMachineBlock stateMachine = addStateMachine("stamp_state_machine", new StampStateMachine(this));
```

What remains to be done is connecting an Ethernet stream from SFP1 to the state machine input and routing the state machine output to the CPU via the PCIe bus:

```
        EthernetStream eth_stream = addEthernetStream(
            "eth_ch2_sfp1",
            Max3NetworkConnection.CH2_SFP1,
            EthernetChecksumMode.DropBadFrames);

        stateMachine.getInput("eth_to_sm") <== eth_stream.getReceiveStream();
        addStreamToCPU("data_out") <== stateMachine.getOutput("stamped_eth_frm_sm");
```

CPU code The application running on the CPU that processes the time stamps sent by the state machine interprets them as a sequence of structures in this form (cf. *Table 9*).

```
typedef struct {
    uint64_t data;
    uint8_t eof : 1;
    uint8_t sof : 1;
    uint8_t mod : 3;
    uint8_t cs_bad : 1;
    uint64_t count : 48;
    uint16_t unused : 10;
} __attribute__ ((__packed__)) eth_stream_t;
```

To record the time stamps in a file, the application converts a sequence of stream words in this form to a sequence of frames. The file format for the frames follows the convention used by utilities such as `tcpdump` and `wireshark`, sometimes known as `pcap` format.[1] Files in `pcap` format have a header at the beginning of the file and a header at the beginning of each frame. The header at the beginning of the file has this form:

```
typedef struct pcap_hdr_s {
    uint32_t magic_number;  // magic number
    uint16_t version_major; // major version number
    uint16_t version_minor; // minor version number
    int32_t  thiszone;      // GMT to local correction
    uint32_t sigfigs;       // accuracy of timestamps
    uint32_t snaplen;       // max length of captured packets, in octets
    uint32_t network;       // data link type
} __attribute__ ((packed)) pcap_hdr_t;
```

The fields in this header are filled with constant values that are not data dependent. The header at the beginning of each frame has this form:

```
typedef struct pcaprec_hdr_s {
    uint32_t ts_sec;        // timestamp seconds
    uint32_t ts_usec;       // timestamp microseconds
    uint32_t incl_len;      // number of octets of packet saved in file
    uint32_t orig_len;      // actual length of packet
} __attribute__ ((packed)) pcaprec_hdr_t;
```

The fields in this header must be calculated by the application.

This calculation is straightforward. Each time a word is read from the stream, its SOF and EOF bits are inspected, and the Data field is written to a buffer whose index is advanced by 8 in each case unless EOF is true, when it is advanced by the number of bytes remaining in the word. The starting count is noted when SOF is true. When EOF is true, the packet header fields are calculated and written to a file

[1] `http://wiki.wireshark.org/Development/LibpcapFileFormat`

followed by the buffer contents.

```
int length = 8;
if (stream_word->eof && stream_word->mod != 0)
    length = stream_word->mod;
memcpy (&(frame[*frame_index]), &(stream_word->data), length);
*frame_index += length;
if (stream_word->sof)
    *starting_count = stream_word->count;
if (!(stream_word->eof))
    return;
latency = ldiv (*starting_count, FREQUENCY);
packet_header.orig_len = *frame_index;
packet_header.incl_len = *frame_index;
packet_header.ts_sec = starting_time + latency.quot;
packet_header.ts_usec = (1000000 * latency.rem) / FREQUENCY;
fwrite ((void *)&packet_header, sizeof packet_header, 1, log_file);
fwrite ((void *) frame, *frame_index, 1, log_file );
fflush ( log_file );
```

The time stamp is determined by the starting count, the frequency of the counter, and the starting time of the application.

The main loop of this application uses the low latency interface described in *section 10*:

```
while(reads < TOTAL_READS)
{
    if (max_llstream_read (incoming_llstream, 1, &stream_word) > 0)
    {
        accumulate_frame (log_file, stream_word, frame_buffer, &frame_index, starting_time, &starting_count);
        max_llstream_read_discard (incoming_llstream, 1);
        reads++;
    }
    else
        continue;
}
```

For performance reasons, a real network application should always use the low latency interface rather than interrupt driven synchronization.

9

Advanced Network Configuration

This chapter summarizes advanced MaxCompiler features pertaining to the network and data link layer, which include IP routing management and ARP functions.

Most of the functions presented here take an `engine` and `connection` as their first two parameters. These parameters have the same interpretation as in the previous chapters and are not discussed any further.

9.1 IP Routing

SLiC supports IP routing and maintains a routing table. Initially, it only contains an entry for the directly connected network (as inferred from the IP address and network mask passed to `max_ip_config()`, see *subsection 4.3*). Without further setup, this allows the DFE to communicate with peers on the local network only.

To enable communication with peers on remote networks a further IP routing information must be provided. A default gateway can be set using the following function:

```
void max_ip_route_set_default_gw(
    max_engine_t *engine,
    max_net_connection_t connection,
    const struct in_addr *gateway);
```

The `gateway` parameter specifies the IP address of a gateway on the local network. The gateway will receive traffic from the DFE that is not destined for the local network and for which no specific route was found in the routing table. Only one default route can be set at a time. Calling `max_ip_route_set_default_gw()` multiple times therefore overrides the previous setting.

Specific routes can be added to the routing table using the following function:

```
void max_ip_route_add(
    max_engine_t      *engine,
    max_net_connection_t connection,
    const struct in_addr *destination,
    const struct in_addr *netmask,
    const struct in_addr *gateway);
```

Similarly to `max_ip_route_set_default_gw()` this function takes a gateway parameter but also requires a destination IP address and a corresponding network mask. Depending on the network mask, the destination may either refer to a network or a specific host (i.e network mask 255.255.255.255). The gateway address 0.0.0.0 has a special interpretation and is used to specify that the destination resides on the local network.

Note that the routing table may only contain one entry for each destination and network mask pair at a time. Routing table entries can be removed or retrieved with the following two functions:

```
void max_ip_route_remove(
    max_engine_t      *engine,
    max_net_connection_t connection,
    const struct in_addr *destination,
    const struct in_addr *netmask);
```

```
int  max_ip_route_get(
    max_engine_t      *engine,
    max_net_connection_t connection,
    const struct in_addr *destination,
    const struct in_addr *netmask,
    struct in_addr         *gateway);
```

For both functions, `destination` and `netmask` should refer to a route previously added using `max_ip_route_add()`. The `gateway` output parameter to `max_ip_route_get()` receives the current gateway address of the specified route.

9.2 IP Multicast

MaxCompiler provides support for IP multicast. IP packets sent to the IP address associated with a multicast group are receivable concurrently by every member of the group. The first function is for joining a group, and the second is for leaving a group.

```
void max_ip_multicast_join_group (
   max_engine_t *engine,
   max_net_connection_t connection,
   const struct in_addr * multicast_ip )
```

```
void max_ip_multicast_leave_group (
   max_engine_t *engine,
   max_net_connection_t connection,
   const struct in_addr * multicast_ip )
```

The `multicast_ip` parameter is the IP address associated with the group. Every network connection is implicitly a member of the All Hosts group, (224.0.0.1). Attempting to leave the All Hosts group has no effect.

9.3 ARP

ARP tables associate IP addresses with physical MAC addresses for peers on the local network, and are necessary for most network applications. SLiC maintains an ARP table for use by the DFE. This table is separate from the ARP table maintained by the operating system (if any).

It is not necessary to manipulate the ARP table explicitly in applications that use UDP or TCP protocols (*section 6* and *section 5*) because use of those protocols initializes the ARP table as a side effect. However, some applications may require direct access to the ARP table, for instance to add static entries to the table.

Entries can be added to the ARP table through the following function:

```
void max_arp_add_entry(
    max_engine_t       *engine,
    max_net_connection_t  connection,
    const struct in_addr     *ip,
    const struct ether_addr *mac)
```

The `ip` and `mac` parameters are the IP address and MAC address of the host on the local network to be added to the ARP table. The entry will be marked as static and therefore remain in the table until explicitly removed.

The following two functions are used to remove or retrieve entries from the ARP table:

```
void max_arp_remove_entry(
    max_engine_t *engine,
    max_net_connection_t connection,
    const struct in_addr *ip)
```

```
int  max_arp_get_entry(
    max_engine_t *engine,
    max_net_connection_t connection,
    const struct in_addr *ip,
    struct ether_addr     *mac)
```

Here, the `ip` parameter identifies a host already present in the ARP table. The `mac` output parameter to `max_arp_get_entry()` receives the MAC address currently associated with the specified host.

10 Low-Latency PCIe Interface

The usual way of communicating between the CPU and the DFE for applications that are not optimized for latency involves the use of hardware interrupts to signal the completion of a transfer. However, interrupts can be inappropriate for low-latency applications because an interrupt generated for each transfer requires a context switch during which the application is suspended while the operating system handles the interrupt. If data transfers are short and frequent, the time it is suspended can become an appreciable fraction of the application's total run time. More critically, it can contribute significantly to the total latency of each operation.

The low-latency interface documented in this chapter saves the time that would otherwise be wasted on these context switches by not using interrupts. This feature is useful when it is important to minimize the time taken to respond to the transfer of data, as it normally is for high-performance network applications.

10.1 Interfaces

The API is designed to allow data transfers to happen concurrently with other processing by the CPU application and Manager blocks, and to do so as efficiently as possible without exposing unnecessary details to the application developer. Kernels, state machines, and Managers may be written entirely without regard for whether this API is in use. It affects only the CPU code, as described in the remainder of this section.

10.1.1 Overview

Because the API affects only the CPU code, this document discusses the transfers from the point of view of the CPU. Hence, transferring data from the DFE to the CPU is identified as reading, and transferring it from the CPU to the DFE is identified as writing.

The CPU application allocates a buffer of adequate size to hold the data in transit for each PCIe stream when using this interface. A separate buffer is needed for each stream, and a given stream can be used only for reading or writing, not both.

Data is transferred in blocks of a fixed size configured at run-time by the developer. Each time a block is queued for transfer, it occupies a slot in the buffer. The slot is released for reuse after the transfer is complete.

The application uses non-blocking API calls to access the buffer, while the hardware independently and concurrently transfers data between the buffer and the DFE.

Users familiar with the concept of a ring buffer may find it helpful to understand the implementation in those terms, but this understanding is not a prerequisite for using the API.

10.1.2 Initialization Functions

An application using this interface starts as usual by calling functions such as `max_load`.

✘ Applications should not call `max_set_pcie_streams_timeout` when using the low-latency PCIe interface, because its effect is unspecified in this context.

Setting up Following the usual initialization, the application calls this function once for each stream:

max_llstream_t ∗max_llstream_setup(
  *max_engine_t ∗engine, **const char** ∗stream_name, size_t num_slots, **const** size_t slot_size, **void** ∗buffer);*

The `max_llstream_setup` function returns a pointer of type `max_llstream_t`, which points to a data structure used internally by the API to track certain information about the state of the stream. The parameters have these interpretations:

- `engine` is an engine handle obtained by a prior call to `max_load`, referring to a DFE used by the application.
- `stream_name` is the name of the low-latency stream.
- `slot_size` is the size in bytes of each unit of data to be read or written atomically. It must be a multiple of 16 ranging from 16 to 4096.
- `num_slots` is the size of the buffer in units of `slot_size` bytes. It may be at most 512. (The buffer size in bytes is therefore `slot_size * num_slots`.)
- `buffer` points to an area of memory aligned on a 16 byte boundary and previously allocated by the application. Its size must suffice to store `num_slots` units of data, each of `slot_size` bytes. This area is used for temporary storage of data in transit.

Freeing a stream When a stream is no longer required, the CPU code should release the resources associated with it by calling this function:

```
void max_llstream_release(max_llstream_t *stream);
```

The `stream` parameter is one that has been returned by a previous call to `max_llstream_setup`. This function does not free the buffer passed to `max_llstream_setup`, which should be explicitly freed separately. It also does not free the stream whose stream handle was previously passed to `max_llstream_setup`, which should be freed afterwards by calling `max_destroy_pcie_streams` or `max_close_pcie_stream`.

Size considerations As far as possible, the sizes should be chosen by the developer with a view to performance and correctness.

- Smaller slot sizes allow more frequent transfers and hence lower latency, but perhaps incur a penalty in throughput due to fixed overheads per transfer.
- Larger numbers of slots improve throughput by allowing either the CPU or the DFE to continue working productively while the other party is temporarily too busy to communicate.
- A further constraint on the sizes derives from the requirement that no slot may straddle a 4096 byte memory address boundary when the buffer is considered as a contiguous array of slots.

A sufficient condition to satisfy the last constraint (albeit stronger than necessary in some cases) is to align the buffer on a slot size boundary and make the slot size a divisor of 4096.

The function `posix_memalign` from the standard C library (`stdlib`) is useful for allocating memory on specifically aligned boundaries.

For example, if a slot size of 256 is chosen, it satisfies the requirements of being a multiple of 16 and a divisor of 4096, and guarantees that no slot straddles a 4096 byte boundary if the alignment is also chosen to be 256, as in this code:

```
posix_memalign ((void *) &buffer, 256, 256 * NUM_SLOTS)
```

The possible slot sizes suitable for this technique are 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096.

10.1.3 Reading Functions

Reading (i.e., transferring data from the DFE to the CPU) involves two steps. First, the application requests a number of slots of data from the stream, and later it releases the slots for reuse.

Requesting new data Requesting the data is done by this function:

```
size_t max_llstream_read(max_llstream_t *stream, size_t max_slots, void **slots);
```

The function always returns immediately, without blocking the application, whether or not any data is available to be read.

- The return value is the number of slots found to contain new data read from the DFE, which may be any number ranging from zero to the `max_slots` parameter.

- If the return value is non-zero, the pointer whose address is passed via the `slots` parameter is modified to point to an array of that number of slots containing the newly read data. (For performance reasons, this array resides within the buffer passed to `max_llstream_setup` rather than the dynamic storage heap, but is accessed by normal C-style pointer dereferences.)
- The `stream` parameter is given by a prior call to `max_llstream_setup`, wherein the `slot_size` parameter determines the size of the slots in the array.

After calling `max_llstream_read` and getting a non-zero result, the application can access the data read from the DFE by way of the pointer `slots` and process it as needed.

Releasing used slots When the data is no longer needed, the application should release the slots by calling the following function:

max_llstream_read_discard(max_llstream_t ∗stream, size_t num_slots)

The storage referenced by the `slots` parameter to `max_llstream_read` is implicitly reclaimed even though no reference to it is passed explicitly to this function. Because the array actually resides within the buffer passed to `max_llstream_setup`, it should not be deallocated any other way.

The `num_slots` parameter to `max_llstream_read_discard` is the number of slots being released, which may range from 1 to the number returned by `max_llstream_read`. If `num_slots` is less than the maximum, then only `num_slots` slots starting from the beginning of the array are released, and the remaining ones can still be used by the application until they are released by subsequent calls to `max_llstream_read_discard`. On subsequent calls, `num_slots` should not exceed the number of slots remaining to be released.

✗ All outstanding slots for a stream being read must be released before any more are requested.

10.1.4 Writing Functions

Writing (i.e., transferring data from the CPU to the DFE) also requires two steps. First the application requests a number of empty slots for it to fill with data bound for the stream, and secondly it requests their dispatch to the stream. The latter step implicitly releases the slots for reuse when the writing is complete.

Requesting empty slots The function to request empty slots for writing is as follows.

size_t max_llstream_write_acquire(max_llstream_t ∗stream, size_t max_slots, ***void*** *∗∗slots);*

The `max_slots` parameter contains the number of slots requested, and the `slots` parameter contains the address of a pointer that is modified to point to an array of empty slots if the call is successful. The return value is a number ranging from zero to `max_slots`, which gives the size of the array in slots. A return value of zero may indicate that the CPU is writing too frequently for the DFE to keep up.

After a successful call to `max_llstream_write_acquire`, the application should fill the slots referenced by the `slots` parameter with new data to be written to the DFE. The writing does not take place until the next step is performed.

Listing 18: Manager connecting the Kernel to the CPU with two streams (LowLatencyManager.maxj).

```
package lowlatency;

import static com.maxeler.maxcompiler.v2.managers.standard.Manager.CPU;
import static com.maxeler.maxcompiler.v2.managers.standard.Manager.link;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;

public class LowLatencyManager {
    public static void main(String args[]) {
        Manager m = new Manager(new LowLatencyEngineParameters(args));
        Kernel k = new LowLatencyKernel(m.makeKernelParameters("LowLatencyKernel"));

        m.setKernel(k);
        m.setIO(link("cpu_to_dfe_stream", CPU),link("dfe_to_cpu_stream", CPU));
        m.build();
    }
}
```

Writing new data The following function calls for new data in the slots acquired by the previous function to be written to the DFE.

```
void max_llstream_write(max_llstream_t *stream, size_t num_slots);
```

- The `stream` parameter is obtained from a prior call to `max_llstream_setup`.
- The `num_slots` parameter ranges from 1 to the number previously returned by the relevant call to `max_llstream_write_acquire`.

If the value passed as `num_slots` is less than the number returned by `max_llstream_write_acquire` previously, only the first `num_slots` slots starting from the beginning of the array are written, and one or more subsequent calls to `max_llstream_write` are needed to write the rest. In the subsequent calls, the `num_slots` parameter should not exceed the number of slots remaining to be written.

The storage associated with the slots used for writing is implicitly released when they are written. No other way of deallocating them should be attempted.

✗ All acquired slots from a stream being written must be released before any more are acquired.

10.2 Low Latency Interface Example

Example 1 demonstrates the low latency interface by transferring buffers full of random data to an DFE, transferring them back again, and comparing the results to the originals. The cycle is repeated several times to confirm proper operation.

10.2.1 DFE Code

The Manager used in this example specifies two PCIe streams, one in each direction, connecting the Kernel to the CPU, and is shown in *Listing 18*.

Listing 19: Loopback Kernel used for demonstrating low latency interface operation (LowLatencyKernel.maxj).

```
package lowlatency;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

public class LowLatencyKernel extends Kernel {
    public LowLatencyKernel(KernelParameters parameters) {
        super(parameters);

        DFEVar StreamIn = io.input(”cpu_to_dfe_stream”, dfeUInt(8));
        DFEVar StreamOut = StreamIn;
        io.output(”dfe_to_cpu_stream”, StreamOut, dfeUInt(8));
    }
}
```

The simple loopback Kernel shown in *Listing 19* is used. This Kernel copies an input data stream with a width of one byte to the output without modifying it. Note that as far as the Kernel is concerned, the streams are read and written the same way as they would be if the low latency interface were not being used.

The low latency interface is not compatible with MAX2 board models.

10.2.2 CPU Application

The CPU code initializes the stream to carry data from the DFE to the CPU like this.

```
    posix_memalign ((void *) &read_buffer, SLOT_SIZE, SLOT_SIZE * NUM_SLOTS);
    read_llstream = max_llstream_setup (engine, ”dfe_to_cpu_stream”, NUM_SLOTS, SLOT_SIZE, read_buffer);
```

The stream in the other direction is initialized similarly.

```
    posix_memalign ((void *) &write_buffer, SLOT_SIZE, SLOT_SIZE * NUM_SLOTS);
    write_llstream = max_llstream_setup (engine, ”cpu_to_dfe_stream”, NUM_SLOTS, SLOT_SIZE, write_buffer);
```

The main loop of the application proceeds as shown in *Listing 20*. Note that this loop polls both streams and performs a read or write whenever conditions allow it. The effect is that reading and writing are interleaved non-deterministically.

A run-time analysis of this example would exhibit an initial transient phase during which a surplus of data originating from the CPU accumulates in the write buffer while relatively few transfers from the DFE to the CPU take place, followed by a steady state in which transfers take place at approximately equal rates in both directions, but not always in strict alternation. Hence there is very little if any waiting time required by either the CPU application or the DFE. This efficient use of time and memory resources makes a design in this style a suitable choice for deployment in many low latency applications.

Listing 20: main loop of the low latency API example (LowLatencyCpuCode.c)

```
completed_reads = 0;
completed_writes = 0;
uint8_t generated_data[TOTAL_TRANSFERS][SLOT_SIZE];
while(completed_reads < TOTAL_TRANSFERS)
{
    if (completed_writes < TOTAL_TRANSFERS)
    {
        // generate more data

        if (max_llstream_write_acquire(write_llstream,1,&write_ptr))
        {
            for (j = 0; j < SLOT_SIZE; j++)
                generated_data[completed_writes][j] = rand() & 0xFF;
            memcpy (write_ptr, generated_data[completed_writes], SLOT_SIZE);
            max_llstream_write (write_llstream,1);
            completed_writes++;
            printf ("Wrote %d/%d\n", completed_writes, TOTAL_TRANSFERS);
        }
    }
    if (max_llstream_read(read_llstream,1,&read_ptr))
    {
        // check data integrity

        if (memcmp(read_ptr, generated_data[completed_reads], SLOT_SIZE))
            status = 1;
        max_llstream_read_discard(read_llstream,1);
        completed_reads++;
        printf ("Read: %d/%d\n", completed_reads, TOTAL_TRANSFERS);
    }
}
```

10.3 Deadlock Avoidance

When code is allowed to execute non-deterministically as in the previous example, it is important for the developer to be mindful of the possibility of deadlock. This section discusses the relevant implications to the low latency PCIe interface.

10.3.1 Deadlock from Deterministic Interleaving

Rather than reading and writing non-deterministically as in the example, a simpler way of accelerating a sequential application that operates on blocks of data would be to transfer each block to the DFE for processing, and then transfer it back to the CPU. The original version of the application might express the relevant operation as a function or subroutine call, which it would seem natural to replace by a function that encapsulates the exchange of data with the DFE. However, this approach will probably fail due to deadlock.

Kernels and other Manager blocks contain pipelines designed for highly concurrent operation. As such, they are generally not expected to empty out until processing terminates. By waiting for a block of data to be returned after sending just one, the application effectively deprives the pipeline of input, resulting in a stall. A stalled pipeline neither sends nor receives. Although the pipeline may contain some results that are finished being computed, they will not be transferred back to the CPU until the pipeline is restarted by being fed more input data, so the application will wait forever.

✗ To avoid deadlock, a CPU application should allow reading and writing to be interleaved non-deterministically.

It is theoretically possible but not recommended to arrange the interleaving deterministically if the pipeline depth is known. The CPU application could wait for the expected amount of data less that which is necessary to fill the pipeline. However, this coding style would impair maintenance because the pipeline depth can be affected in non-obvious ways by minor changes to the source code, or by future changes to the compiler.

10.3.2 Deadlock from Partial Transfers

Data is transferred by this interface only in multiples of the slot size. If a Kernel transfers less data to the PCIe bus than needed to fill a whole slot, the hardware waits indefinitely until it transfers enough to fill the rest of the slot. If a Kernel reaches the end of its run and transfers less than the amount needed to fill the last slot, the CPU code will poll forever because `max_llstream_read` will never return a non-zero value. This cause of deadlock can be prevented in applications where the number of bytes to be transferred is known in advance.

✗ To avoid deadlock, the total number of bytes transferred should be a multiple of the slot size.

If unpredictable transfer sizes make the guideline above difficult to follow, padding the data up to the slot size is a valid alternative. This solution requires CPU applications and Kernels to agree on a rule for indicating the amount of useful data in a slot. An unsigned integer at the end of each slot indicating its length is a good choice. Another choice is to append a pattern to the data that is known never to appear within it, which signals the beginning of the pad.

SLiC API Index

MaxJ API Index