Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

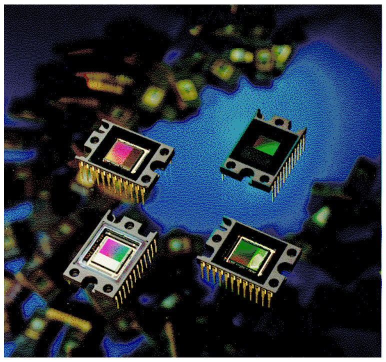
2003

MSc THESIS

TTL Interfaces for Multiprocessor Platforms

Yanning Luo

Abstract



The key issues in the design of Systems-on-Chip (SoC) are efficiency, design cost and time to market. To deal with the trade-off between them, the idea of component standardization has been adopted. In Philips Research, a design team is working on a standard multiprocessor interface called Task Transaction Layer (TTL). TTL is basically an application programming interface (API) that allows the Intellectual-property (IP) blocks to interact with each other in a uniform way such that the design of the system functionality can be separated from the design of architecture and the two parts can still be easily integrated in a plug-and-play fashion. The initial focus is on streaming-based multimedia processing embedded systems. During the TTL design phase, different design choices on inter-task communication have been proposed. The main task of this thesis is to evaluate those design choices in the aspect of ease of programming in a quantitative way. To achieve this, a top-bottom design methodology: Y-chart, typical multimedia processing applications and the proposals of TTL are studied, an evaluation method is proposed and applied on the comparison of TTL design choices. The result can be used as a reference in TTL design choice making as well as a guide in TTL application design.

CE-MS-2003-04

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

TTL Interfaces for Multiprocessor Platforms

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Yanning Luo
born in Shanghai, China

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

TTL Interfaces for Multiprocessor Platforms

by Yanning Luo

Abstract

The key issues in the design of Systems-on-Chip (SoC) are efficiency, design cost and time to market. To deal with the trade-off between them, the idea of component standardization has been adopted. In Philips Research, a design team is working on a standard multiprocessor interface called Task Transaction Layer (TTL). TTL is basically an application programming interface (API) that allows the Intellectual-property (IP) blocks to interact with each other in a uniform way such that the design of the system functionality can be separated from the design of architecture and the two parts can still be easily integrated in a plug-and-play fashion. The initial focus is on streaming-based multimedia processing embedded systems. During the TTL design phase, different design choices on inter-task communication have been proposed. The main task of this thesis is to evaluate those design choices in the aspect of ease of programming in a quantitative way. To achieve this, a top-bottom design methodology: Y-chart, typical multimedia processing applications and the proposals of TTL are studied, an evaluation method is proposed and applied on the comparison of TTL design choices. The result can be used as a reference in TTL design choice making as well as a guide in TTL application design.

Laboratory : Computer Engineering
Codenumber : CE-MS-2003-04

This thesis is dedicated to my parents

Contents

List of Figures

List of Tables

Acknowledgements

This master thesis is supported by the Computer Engineering Laboratory at Delft University of Technology (TUDelft) and the Embedded System Architectures on Silicon (ESAS) group in Philips Research. As an Msc Student from TUDelft, I consider this as a great fortune to work in such a professional environment, with which I have got both academic and research industrial experiences.

First and foremost, I would like to thank Prof. Vassiliadis and Pieter van der Wolf for giving me the opportunity to work in their groups. I also want to thank all the people who have been involved in the administrative work for making this happen.

I am very grateful to Koen Bertels, my thesis supervisor from TUDelft, who gives me many important advices on my thesis work. Following his suggestion of writing reports while working on the project has kept me refreshed on every step of the project and saved me from a rush at the end. His advices on the structure of my thesis not only led to a big improvement of the thesis but also opened my eyes on English report writing.

I want to say thanks again to Pieter van der Wolf, my supervisor from Philips Research, who has guided me during the nine months. His insights to the research in the area of multiprocessor platforms and his commitment to research work have largely inspired me.

Certainly I want to express my deep gratitude to Gerben Essink, who has given me very strong technical support all along the development of the project and the thesis. For so long, I have been impressed by his knowledge and patience.

I am very thankful to Jeffrey Kang, who has so carefully checked my thesis draft and provided valuable advices. He is also the one who often encourages me to take more initiatives while working in a Dutch company.

I also want to acknowledge Erwin de Kock for his support on the runtime environment setting up and my fellow student Bao Linh Dang for his great contribution on the template of the thesis and his help on Latex.

I was able to enjoy the work partly because I have a very active international student group around. I am very glad to have met each of them and shared so much fun time together. Especially thank David Smola and his family for the lovely winter break in Czech Republic.

Special thanks are for my family and my boyfriend. I want to thank my father who passed away four and a half years ago. I want to thank him for all his love and education which will definitely be beneficial all through my life. I am also indebted to my mother for her selfless love and feel deeply sorry for all her lonely time at home. Finally I would like to thank my boyfriend, Frank Vossen for his support at all my up and down time.

Yanning Luo
Eindhoven, The Netherlands
July, 2003

1 Introduction

This document is the report for TTL ease of programming study. The TTL ease of programming project serves for the Task Transaction Level(TTL) Interface project in Embedded System Architectures on Silicon(ESAS) Group, Information and Software Technology(IST) Sector, Philips Research. TTL is a unified platform interface for multi-processor Systems-on-Chip (SoC). The overall goal of the TTL project is to define an API targeting multiple platforms that allows different application tasks to communicate with each other in an uniform way. In this way, both the applications and the architectures can be reused for other systems. So far, the framework of the TTL specification has been completed, in which the main categories of services that TTL is going to support are defined. The design team is currently working on the detailed design choices that are to be made under each category. The goal of the TTL ease of programming study project is to evaluate these design choices in the aspect of design effort from the application designers' point of view. To achieve this, we propose a method to assess the design effort required from different design choices in a quantitative way and apply them on the TTL inter-task communication decision making. With the results of the project, the TTL design team can have a clear view of the application designers' preferences concerning the ease of programming. The method proposed in this project can be further used to evaluate other design choices in other embedded system interfaces. The code example given in this document can be referred to as a programmers' guide for TTL application designers as well.

The organization of the thesis is as follows: The background information of complex systems-on-chip design is given in Chapter 2. In Chapter 3, the problem definition of the project is presented. We describe the related work in data communication domain and task interaction in embedded system domain in Chapter 4. We also look at how other teams implement inter-task communication in their real-time infrastructures in the same chapter. The methodology we are going to use is proposed in Chapter 5. According to the proposed approach, we study the characteristics of the inter-task communication and collect typical application code examples in multimedia processing and introduce the evaluation criteria and weight factors in Chapter 6. In Chapter 7, we look into the TTL design choices and propose a comparison strategy for the evaluation. In the following five chapters(8-12), we assess the TTL design choices based on the comparison strategy. We conclude the whole thesis in Chapter 13.

Design of Complex Systems-on-Chip 2

As mentioned in the previous chapter, the purpose of this chapter is to give some background information about the design of complex systems-on-chip. In Section 2.1, we discuss system design methodology. In Section 2.2 and 2.3, two important steps in system design will be discussed. In Section 2.4, we will discuss the Task Transaction Level interface.

2.1 System Design Methodology

The continuous advent and development of new applications, especially media applications, demands more complexity of systems-on-chip(SoC). As a result, modern embedded systems often have a heterogeneous architecture, which consists of fully programmable processors and application-specific subsystems. With the requirements of performance, time-to-market as well as cost and power effectiveness, classical design methods are facing threats in mainly two aspects:[4]

- Classical design methods that start from a single-application specification and gradually synthesise into an architecture implementation usually make the system dedicated to only a single application and thus give low reusability to both hardware and software application components and the architecture.
- The growing complexity of the systems-on-chip architectures brings more difficulties and more necessity to predict the performance behaviour at an earlier design stage.

All these call for a change in design methodology for embedded systems. An important step is to separate the system's function design from the architecture design. Here we use the system design Y-chart[15] to visualise the approach, as shown in Figure 2.1[15].

The figure above distinguishes four activities: application modelling, architecture modelling, mapping and performance analysis. Typically when doing application modelling, the application designers study the function specification of the system, define a set of benchmark applications and describe the functional behaviours, including both computation behaviours and communication behaviours. Note that the application is not software nor hardware, it is a description of functionality that can be implemented in partly software and partly hardware depending on the target architecture it will be mapped on at a later stage. During architecture modelling, the system designers study the set of benchmark applications, make some initial calculations and propose candidate architectures. An architecture defines the resources available in the system and

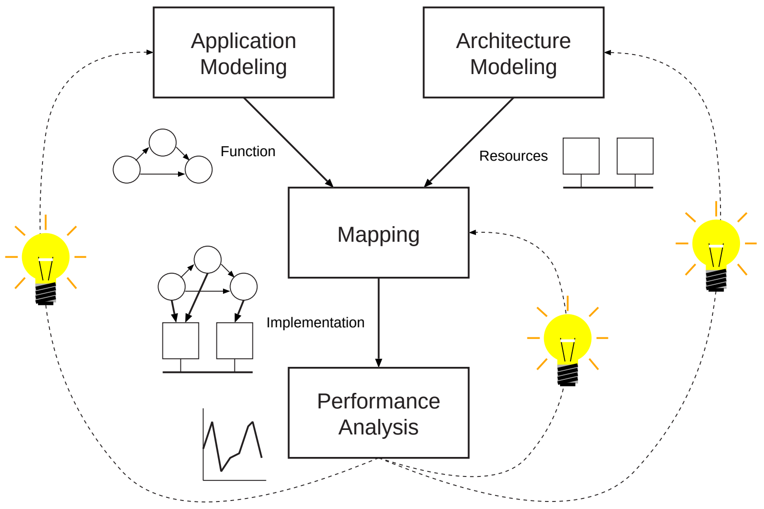


Figure 2.1: System design Y-chart

the performance constraints. Typically for embedded systems the resources are processors, coprocessors, operating systems, buses, memories, time and power. The function is then mapped onto each candidate architecture design, the result of which is an implementation of the system. During performance analysis, the results of mapping and the application-architecture combinations are evaluated quantitatively against each other.

This modelling methodology keeps the application modelling independent of architecture modelling so that a single application model can be reused on different architecture models and an architecture can be reused for different function implementation as well. Furthermore, this methodology also facilitates the simulation because designers can simply run the application on different architectures.

2.2 Application Modelling

To capture the functional specification of a system, we need a program interface that can explicitly describe the functionality, distinguish between the computation and communication behaviour as well as enable modular construction and reuse such that the mapping on different architectures will be easy. A good example of such a program interface is YAPI[6][7], standing for Y-chart Application Programmer's Interface.

In YAPI, the application modelling is based on Kahn Process Networks (KPNs)[14]. From an application designer's point of view, a KPN consists of concurrent processes that communicate with each other via unbounded fifos. The network ensures a deterministic behaviour, i.e. the same input always results in the same output. A process can be merged into a process network without a specification of execution order for the processes. A visualised process network example is shown in Figure 2.2.

In Figure 2.2, a process is represented by a circle, a port is represented by a black dot and a fifo is represented by an arrow. Each process has its own private state space

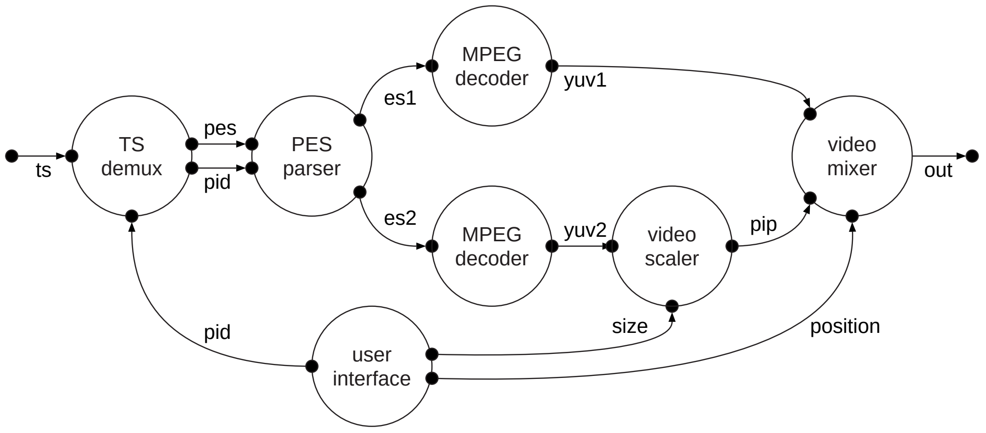


Figure 2.2: A Process Network

which is inaccessible by other processes and a process can only communicate with its environment via its input and output ports. Each fifo has exactly one input port and one output port and they are all associated with a same data type.

YAPI serves as a C++ library so that the computation behaviours of a process can be described by standard C++ and the communication behaviour can be modelled by the primitives provided by YAPI. Such primitives are for example:

```
template<class T> void
read(Inport<T>&s, T& t)
```

The function reads a value from input port s and stores the value in variable t.

```
template<class T>
void write(Outport<T>& s, const T&t)
```

This function writes the value t to output port s.

A simple producer-consumer example of a YAPI program is presented in Appendix A. In the program in Appendix A, the process network PC contains two processes: producer and consumer, and the structure of PC is shown in Figure 2.3. The producer writes data to the fifo, and the consumer reads the data from the fifo.

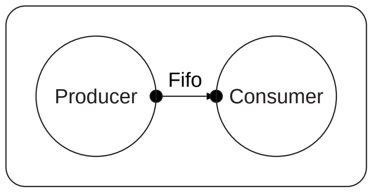


Figure 2.3: producer-consumer

Such an application design separates the computation behaviour from the communication behaviour and leaves the architecture related implementation details to architecture modelling and mapping. Thus an application designer only needs to concern the functional description and does not have to care about whether the functions are going to be implemented in hardware or in software.

2.3 Mapping

In the phase mapping, depending on the different choices made in architecture modelling (processors, operating system, buses, memories and etc.), different applications/IP blocks can be implemented on software or hardware. Examples of such choices are images running on different platforms and different CPUs or coprocessors, accelerators. The fifos between the processes are usually mapped onto buffers/memories in a system.

To provide a uniform interface to different existing program interfaces, we introduce a new interface. This new interface is called Task Transaction Level interface. With the TTL interface, an application/IP block can interact with its environment in the same form, regardless of the different platforms it might be planted into. Thus a reliable plug-and-play functionality is achieved.

2.4 Task Transaction Level Interface

In order to decouple the application design and the architecture design, we should split a system implementation into a platform-dependent part and a function-dependent part so that the components of each part can be reused. That is, a functional block implementation (eg. a software code which is written for some function such as MPEG decoding or an application specific processor for DCT) can be mapped and reused on other platforms and a platform can be adopted by other applications as well. The system implementation can be viewed as shown in Figure 2.4.

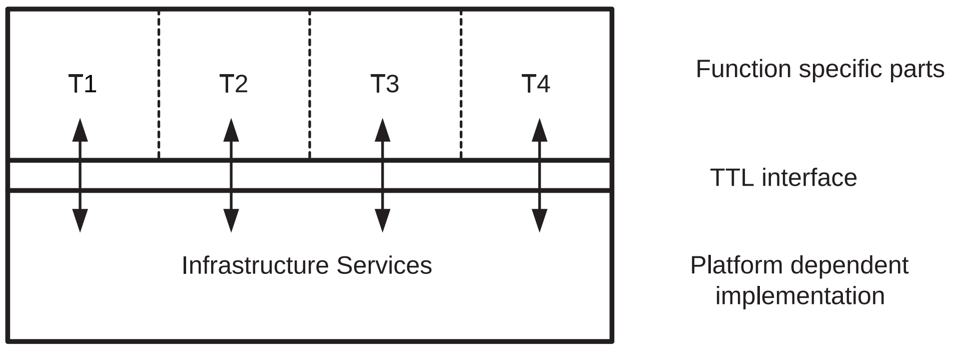


Figure 2.4: System Implementation

The platform-dependent part is usually about how communication with other parts of the system is implemented. We call it the communication kernel. It offers communication services to the function specific parts. The function specific part is called the computation kernel. The implementation of the communication kernel may be specific for a particular platform instance, but it can give the same interface to the computation kernel as other implementations do on other platforms. The uniform interface is the TTL interface. The computation kernel can then interact with the environment by means of the TTL interface. In this way, the implementation details of the communication kernel can be hidden to the computation kernel and the computation kernel can be reused on other platforms. Note that the TTL interface is an abstract interface that can have both hardware and software implementations. In this setup, hardware and software can be integrated in a coherent way. Furthermore, the communication kernel must be implemented in a modular and parameterised way to make it more generic and easier to be reused.

The TTL applications use the functions provided by TTL to talk to the environment. The infrastructure provides services to the function specific parts via the TTL interface. We call these services TTL services. Note that TTL services on different platforms have the same interface, but not the same implementation. TTL only defines a standard interface for the infrastructure services, not a standard implementation of the services.

The whole structure of a system can be viewed as in Figure 2.5. Here the corresponding system level design steps are also indicated. The result of mapping is an implementation of the system.

The services that the TTL offers can be categorised into the following classes:[10]

- Data transfer services: virtual addresses/no addresses, latency hiding (e.g. by means of prefetching), guaranteed throughput, multi-cast
- Synchronisation services: inter-task synchronisation, fifo based synchronisation
- Application reconfiguration: starting and stopping of tasks, parameter setting for tasks
- Task scheduling: support for multi-tasking IPs
- Memory management service: allocation and resizing of shared fifo buffers, non-fifo memory services

These services are accessible by means of primitives. Typical primitives that will be offered by TTL are for example:[10]

Producer's side data transfer:

`acquireRoom(channelId, ...)`: Begin the output of a value on the channel. After finishing the operation, sufficient output space is available to store the value.

`storeData(channelId, ...)`: Copy data from the task to the channel.

`releaseData(channelId, ...)`: End of the output of a value on the channel. This operation is used to indicate that the value has been produced.

Consumer's side data transfer:

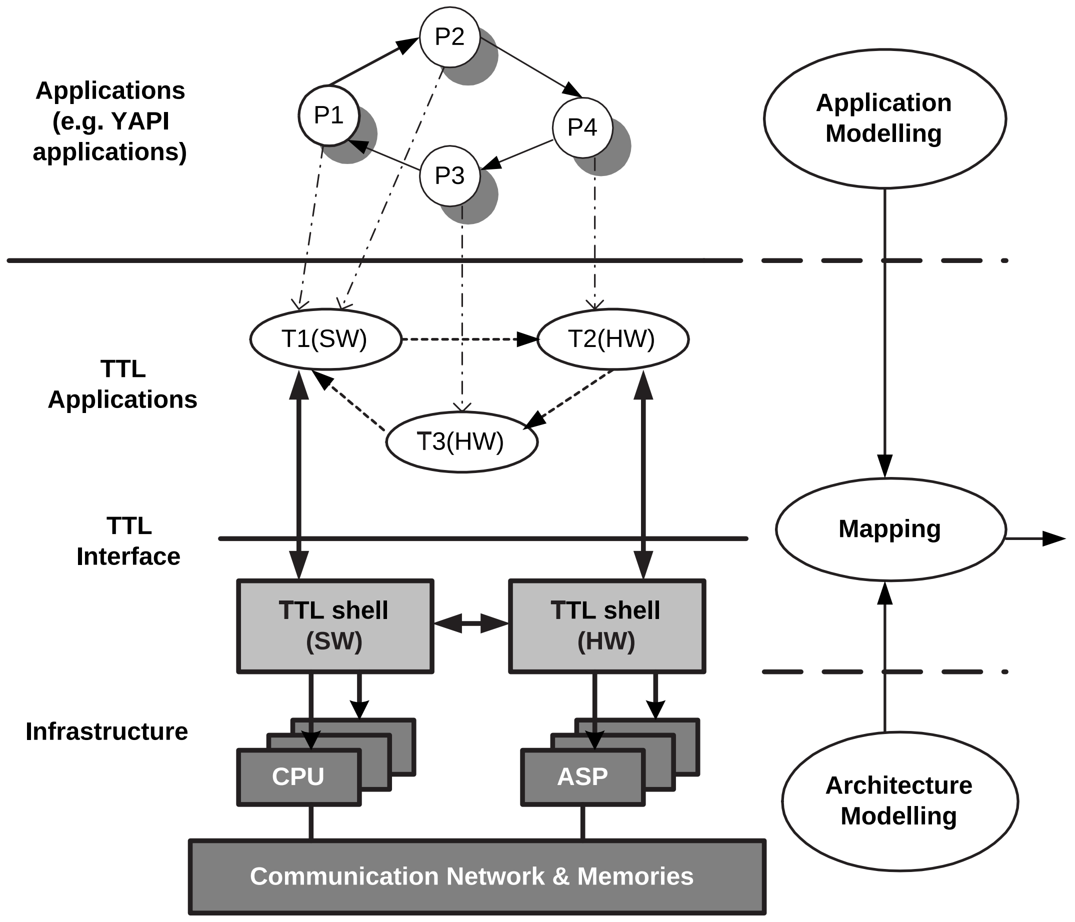


Figure 2.5: System Structure

`acquireData(channelId, ...)`: Begin the input of a value from the channel. After finishing this operation, the input data is available to be loaded.

`loadData(channelId, ...)`: Copy the value from the channel to the task.

`releaseRoom(channelId, ...)`: End the input of a value from the channel. This operation is used to indicate that the value has been consumed.

Application reconfiguration:

```
createTask(taskName, ...)
createChannel(channelName, taskId, portId, taskId, portId, ...)
```

Infrastructures implement the TTL services that are used by applications. At the same time, TTL primitives have corresponding primitives in different program interfaces. In this way a YAPI application can be translated to a TTL application as shown in the example below:

YAPI application fragment:

```
while (1)
{
    for (i=0; i<100; i++)
    {
        // read a token value from input port into
        // a local variable t.
        read( in, t );
        // f(t) is a computation function which is
        // defined in other part of the program
        y = f( t );
        // write the value of y to output port
        // out.
        write( out, y );
    }
}
```

A corresponding TTL application translation can be (TTL1):

```
while(1)
{
     for (i=0; i<100; i++)
      {
          // Acquire one full tokens from channel in.
          acquireData(in );
          // Acquire one empty tokens from channel out.
          acquireRoom(out);

          // Copy the value of the token with distance 1
          // to the oldest acquired token into the
          // variable t.
          load (in, 1, t );
          // Release an empty token to channel in.
          releaseRoom(in );

          y=f(t);

          // Copy the value of y into the token with
          // distance 1 to the oldest acquired token in
          // channel out.
          store (out, 1, y );
          releaseData(out);
      }
}
```

If the implementation is software, the TTL application will be like the program above. If it is hardware, the TTL application will be a hardware block that is connected to the hardware TTL interface.

Here we emphasize the differences between an application interface e.g YAPI and the TTL interface. An application interface is a function description which makes no assumptions on the implementations, whether they will be software or hardware. Rough estimates of the performance can be obtained, but only at a very abstract level. The focus of the application modelling step is on the functionality. However, at the level of TTL, software-hardware partition has been done and simulation can already be performed based on cycle-accurate models. To meet the requirements of performance is thus the main concern of the step mapping. However sometimes an improvement on performance means a comprise to reusability. Hence to provide a high performance and at the same time still maintain a high reusability is the goal of the TTL interface and application design.

As in the above example, the reason why TTL supports separate synchronisation and data transfer (using acquireData/Room, load/store, releaseRoom/Data instead of read/ write) is because in a multiprocessors with shared memory address space, this method usually provides higher efficiency.[18] A more detailed explanation is given in the chapter of TTL design choice study.

2.5 Conclusions

This chapter provided some background information on the new trend of complex systems-on-chip design. We first introduced the Y-chart system design methodology which proposed an idea of decoupling application design and architecture design such that both the function specific parts and the platform specific parts of a system can be reused on other systems and performance analysis can be done at different abstract levels. Then two main activities in this modelling methodology: application modelling and mapping, which are most related to the TTL were explained in detail. Finally, we defined that the TTL is a standard platform interface that allows the application components and the platforms communicate in a uniform way. With the TTL, the reuse of the function specific components and the platforms can be achieved in a plug-and-play fashion. In the next chapter, we will present the problem definition of the project.

3 Problem Definition

The design of complex systems-on-chips were briefly introduced in the previous chapter. We showed that by using the Y-chart system design methodology and a standardised platform interface, e.g. TTL in SoC design, we can largely reduce the design cost, time to market and still meet the requirements to performance. In this chapter, we discuss the goals of the TTL ease of programming study project and how it can contribute to the TTL project. We start with an example of the TTL design choice evaluation in Section 3.1. We describe the scope of the project in Section 3.2 and finally conclude the problem statement and the objectives of the project in Section 3.3.

3.1 TTL Design Choice Evaluation

According to the different design choices made for TTL primitives, the TTL applications may be implemented differently. Those choices are for instance between blocking and non-blocking synchronisation, low-level communication data types, combined vs. separated synchronisation and data transfer, in-order vs. out-of-order data transfers. . . A very simple example of a choice can be shown with the code fragments above. In the previous code example, we called acquireData/Room and releaseRoom/Data for each token (named single token synchronisation or scalar synchronisation) inside the loop. We can also call acquireData/Room, releaseRoom/Data out of the loop by acquiring and releasing the whole piece of room and data (called multi-token synchronisation or vector synchronisation) and processing them token by token in the loop, as presented in the following. With vector synchronisation, the synchronisation frequency can be reduced.

Another TTL application translation from a different choice (TTL2):

```
while(1)
{
      // Acquire 100 full tokens from channel in.
      // (vector synchronisation)
      acquireData(in, 100);
      // Acquire 100 empty tokens from channel out
      // (vector synchronisation)
      acquireRoom(out, 100);

      for (i=0; i<100; i++)
      {
          // Copy the value of the token with distance i
          // to the oldest acquired token into the
          // variable t.
          loadData(in, i, t);

          y = f(t);
```

```
            // Copy the value of y into
            // the token with distance i to the oldest
            // acquired token in channel out.
            storeData(out, i,  y);
        }
        // Release 100 empty tokens to channel in.
        // (vector synchronisation)
        releaseRoom(in, 100 );
        // Release 100 full tokens to channel out.
        // (vector synchronisation)
        releaseData(out, 100 );
   }
```

In order to evaluate different choices for the TTL interface design, the following aspects need to be taken into account:

- Performance in clock cycles
- Ease of programming/design effort

Performance in clock cycles:
We can measure the performance of different applications running on different platforms by calculating and comparing the number of clock cycles each mapped TTL application needs. For instance, for the above two TTL translations running on a certain platform, the cycle number of each operation is listed as follows:

	TTL1	TTL2
acquireData/Room	10	20
load/storeData	10	10
y = f(t)	40	40
releaseData/Room	10	20

Table 3.1: an example of cycle numbers for each operation in two TTL applications

Then we can obtain the number of clock cycles for each TTL application:

TTL1:

$$NC1 = 100 \times (10 + 10 + 10 + 10 + 40 + 10 + 10) = 10000 \quad (3.1)$$

TTL2:

$$NC2 = 20 + 20 + 100 \times (10 + 40 + 10) + 20 + 20 = 6080. \quad (3.2)$$

where $NC1$ and $NC2$ are the numbers of of cycles required from TTL1 and TTL2 respectively.

In this case, TTL2 has better performance because it needs less clock cycles. It can also happen that on some other platforms, the scalar acquireData/Room and releaseData/Room (like in TTL1) are much cheaper in clock cycles than vector acquireData/Room and releaseData/Room (like in TTL2) so that it is more efficient to use TTL1.

Ease of programming:
Besides the issue of performance, we also need to compare the design effort for different cases. For instance, in the above example, TTL1 is a more straightforward translation than TTL2.

The evaluation of the design effort also includes the assessment of the cost of implementation and the reusability of the application code. A low cost of implementation means an ease of programming when the application is created. A high reusability means that little design effort is required each time when the application code is used in a new system. A detailed study of the ease of programming criteria is shown in Section 6.3.

Through the ease of programming study, we expect to have a clear view of the preferences from application designers, which as an important factor will be considered when the final TTL design choices are made.

3.2 Scope of the Project

There are still many design choices to be made in making the specification for TTL primitives. The main issues can be categorised into four classes: [10]

- Time: issues concerning whether there is a notion of time in TTL, and if there, which model of time will be used
- Tasks: issues related to scheduling, task switching, state saving...
- Inter-Task Communication: issues concerning channels, communication data type and synchronisation
- Configuration: issues concerning the network structures, ports, creation and destruction of tasks

Inter-task communication is the part that is most frequently used and has most interfaces with application designers. How to define and choose communication primitives in order to facilitate both the application design and the TTL primitive implementation is one of the most important issues in the TTL design phase. In this thesis, we will study the design proposals for TTL inter-task communication and analyse the ease of programming aspect from an application designer point of view.

3.3 Conclusions

We summarise the problem statement of the project as follows: The increasing complexity of SoC design requires the support of reliable plug-and-play and reusability of IP blocks. This can be achieved by providing a standard interface between IP blocks and multiprocessor platforms. The project Task Transaction Level Interface is to define such a platform interface so that the implementation of an IP block is independent of the implementation of the platform it might be planted into. The project TTL Ease of Programming Study is to propose a method to evaluate the design choices the TTL project has in its early phase in the aspect of design effort, apply the method on the TTL inter-task communication interface design choices and provide suggestions on the decision making according to the results of the evaluation. The initial focus is on streaming based multimedia processing applications.

Our objectives are: Given a set of proposals on TTL inter-task communication interface definition, discuss which proposed interfaces are easier to use from an application designer point of view. The results of the study together with a performance study can be used as a reference in defining the TTL specification and in mapping YAPI applications to TTL applications.

In order to have a broader view and a deeper insight to the research problem, we will study some related work in the areas of data communication and task interaction in embedded systems. In the next chapter, we will describe the main networking techniques in both areas and show some successful realtime infrastructure examples.

4 Related Work

As we describe in the earlier chapter, TTL is an interface for multiprocessor platforms with its initial focus is on streaming-based multimedia applications. Its goal is to define an abstract interface that will facilitate both the application and platform design. Though the implementation details are left to the platform designers, the study of communication mechanisms and implementations in other teams' related work will certainly be beneficial. In this chapter, we first describe basic networking strategies in data communication domain and inter-process communication in embedded system domain. Then the work of Ptolemy, VxWorks, pSOSystem and Occam is discussed. We also give some motivations on the TTL choices. Finally we conclude this chapter in Section 4.4.

4.1 Data Communication

In data communication domain, switching strategies can mainly be categorised into circuit switching and packet switching.[3] Circuit switched networks require connection establishment. All packets are delivered in correct order. However packet switched networks use connectionless transmission.[16] A data stream is split into a number of packets and all the packets are sent and traverse in the network without any interrelation.[9] The characteristics of circuit switching make it more suitable for real-time applications because it provides guaranteed throughput as a result of which a delay bound is also assured. This is especially important to multimedia applications due to the high bandwidth data flow. Packet switching is harder to implement real-time communication since there is no time relation between packets to guarantee the original data stream behaviour. It would also cost too much overhead to use packet switching for multimedia applications because every packet can only contain limited number of bytes, but for each packet, information like destination address, packet length has to be attached, which will lead to a degrade of the bandwidth. Nevertheless, it is perfect for busty behaviour like Internet access since no "setup phase" is required and resources can be shared by many connections.

4.2 Task Interaction in Embedded Systems

Besides the data communication among hosts, tasks running on an embedded system also need to interact with each other. There are three types of interactions possible: communication, synchronisation and mutual exclusion.[19] Communication is simply used to transfer data between tasks. Synchronisation is used to co-ordinate tasks and mutual exclusion is used to control access to shared resources.

There are mainly two communication mechanism categories, i.e. channels and pools. Channels provide the medium for items of information to be passed between one task and another in an ordered manner. Channels can be implemented with queues (commonly FIFO), circular queues, event flags, sockets and pipes. Pools act as a repository of information: items of information are available for reading and/or writing but do not flow within a pool. Pools can be implemented with mailboxes or monitors. Considered the characteristics of streaming-based applications, TTL uses channel communications. Examples of the implementations in other projects are given in the next section.

Communication is based on either shared-memory or message passing. With shared memory-based communication, each task may access or update pieces of shared information/data. With message passing-based communication, a direct transfer of information occurs from one task to another. Shared memory-based communication is used in multiprocessors with a shared memory address space. A message-passing scheme is suitable for multiprocessors with multiple address spaces. In the latter case, the destination processor receives a message either by means of polling or interrupts.[13] A message passing scheme creates copies of data, which requires more bus transfers and memory. When a shared memory space is implemented, a shared memory scheme avoids this by mapping the FIFO in the memory location which is shared by both processes. Hence, tasks can alternatively access the data after claiming the ownership of the data. The act of claiming is synchronisation.[2]

Synchronisation involves the ability of one task to stimulate or inhibit its own action or that of another task. It can be seen as two procedures: wait and signal. One task must wait for the expected event to occur and the other task will signal that the event has occurred. These can be implemented by using semaphores.

Mutual exclusion is used to protect shared resources. It is usually implemented also via semaphores.

4.3 Inter-task Communication in Real-time Infrastructure Examples

Having studied the basic communication methods, below we discuss some successful real-time infrastructures, where these methods are used.

Ptolemy[1] is a software infrastructure (kernel) as well as an environment developed by Berkeley, which supports modelling, simulation, design and code generation for concurrent, embedded real-time systems. The focus is also on systems that mix a wide range of operations, such as signal processing, feedback control, sequential decision making and user interfaces. In order to handle such complexity and at the same time still optimise every subsystem, a variety of computation models are supported. Each of them has a particular strength in dealing with a certain kind of systems, e.g. Communicating Sequential Processing (CSP) is very good at resource management; Process Networks (PN) matches signal processing and Finite-State Machines (FSM) is excellent for control logic. In Ptolemy, those computation models can be organised as a hierarchical system by a user, where a component of a system complying with one model can be a subsystem complying with another model. The way a component interacts with another differs in each computation model, in order to suit the characteristics of each model. But they can be mainly categorised into four mechanisms: mailbox communication, asynchronous message passing, rendezvous communications, discrete-event communications. A mailbox is a receiver that has capacity only for a single token. Asynchronous message passing here is also based on Kahn process networks, where communication is done via FIFOs, as in TTL. Rendezvous Communication or synchronous communication requires both the peers to be ready before a data transfer can take place, which means a producer process will be suspended until the consumer process reaches the synchronisation point to exchange data with an atomic operation. In the discrete-event(DE) model of computation, processes communicate by sending tokens across connections. When a token is sent, it is packaged with the time at which it is sent to form an event and is stored in the global event queue in which events are sorted based on their time stamps and their destinations. A DE domain scheduler ensures that events are processed chronologically.

Ptolemy supports a much broader range of applications than TTL which has its focus mainly on streaming-based data processing. An equivalent domain is the PN, where FIFO channels are also used as the main communication mechanism as in TTL. Inter-processor communication in Ptolemy is based on message passing instead of shared memory. To send or receive a token, a process calls the `get()` or `send()` methods, which take care of both synchronisation and data transfer. Thus it is easy to use.

Another interesting infrastructure is VxWorks[8][17]. VxWorks is an interactive development environment and real-time kernel. It provides multiprocessing communications in different levels. The kernel uses and provides shared memory and semaphores for the synchronisation and interlocking of tasks; message queues and pipes for intertask message passing within a CPU; shared memory objects, i.e. shared semaphores, shared memory queues and shared-memory partitions for intertask communication across processors; sockets and remote procedure calls(RPC) for network-transparent intertask communication. With sockets, communication among processes can be done exactly the same regardless of the location of the processes in the network, or the operating system under which they are running.

Among the inter-task communication mechanisms VxWorks provides for multiprocessor platforms, shared memory objects are difficult to use because the tasks have to take care of the mutual exclusion and synchronisation. Message queues combine synchronisation and data transfer. Sending or receiving a data is done by simply calling the `msgQSend()` or `msgQReceive()` primitive. VxWorks message queues support multiple producers and consumers. Multiple tasks can send to and receive from the same message queue. TTL has decided to support only single producer channels because of the streaming data pattern. Pipe interfaces are similar to those of message queues, only they are meant to communicate with VxWorks I/O systems and one more primitive `select()` is provided to allow a task to wait for data to be available on any of a set of I/O devices. So they are also easy to use. The use of sockets is rather complicated. To establish a socket communication with TCP, it takes the sender task up to five steps/primitives and the receiver seven steps/primitives. However, the remote procedure calls are designed to be user-friendly. Before making any RPC-related calls, a task simply needs to call the primitive `rpcTaskInit()`.

VxWorks supports communication primitives from a low level, sharing of data to a high level, network based intertask communication. TTL is an abstract multiprocessor platform interface for streaming based applications which require an ordered communication service. On one hand, low-level communication services such as shared memory do not provide an ordered service and the use of them often requires a combination with other communication services like semaphores. TTL has decided not to support low-level communication primitives. The techniques of the low-level communication mechanisms can however be used to implement the high-level level streaming services.[10] On the other hand, as a multiprocessor platform interface, TTL does not need to provide a network based level of communication. The use of such a high-level transparent communication will only comprise the efficiency because system software must do more work.

pSOSystem[12] is a modular, high-performance real-time operating system designed for embedded microprocessors. It makes no assumptions about the execution/target environment. The hardware related parts are realised by configuring the parameters of the system components at start-up or provided by the users as separate components which have a standard interface with the system components. pSOSystem supports multitasking on both single and multiple processor systems. To perform task-to-task or ISR-to-task communication, synchronisation and mutual exclusion within one processor, three sets of facilities are provided. They are message queues, events and semaphores. Message queues serve as a general communication mechanism in pSOSystem. Data transfer can be done via message queues. Events make it possible for a task to wait for one signal, one of several or all of several signals by checking the bit-wise event flags of each task, but no data can be carried. Semaphores are mainly used as resource tokens in implementing mutual exclusion. Besides all the synchronous communication mechanisms, pSOSystem also provides asynchronous signals, which allows the sending task to force the receiving task to enter its user-defined Asynchronous Single Service Routine(ASR), where the signal received is handled. Inter-task communication across processors in pSOSystem is supported by means of remote service calls(RPC). Upon an RPC event, the pSOS+m kernel (multiprocessor multitasking kernel) on the source and destination nodes call their respective user-provided communication layer named the kernel interface(KI) to exchange packets. KI has a standard interface with pSOS+m kernel. In this way, the inter-processor communication is completely transparent to the applications.

The interfaces of pSOSystem inter-task communication are comparable to those from VxWorks. A message queue in pSOSystem is created dynamically and accessed by tasks using `q_create`, `q_send` and `q_receive` system calls. A queue is not explicitly bound to any task, therefore serves as a many-to-many communication switching station, which is different from TTL. An identical set of primitives are defined for semaphores as in VxWorks. Two special features here are events and asynchronous signals. Two system calls are provided for event operation: `ev_receive` and `ev_send`, which are used to get or send events from or to a task respectively. The use of asynchronous signals takes at least three system calls: `as_catch` to establish a task's ASR, `as_send` to send signals to a task and as_return to return from the ASR.

pSOSystem provides a wide range of operating system services for real-time systems. However, for some systems, such a complete set of service is too costly. In order to suit a variety of systems, TTL has to define a more flexible platform interface. As a result, a set of more abstract interface primitives is required. pSOSystem uses message passing for inter-process communication, which has the drawback of having to copy data.

A model that has the design principle of ”keep things simple” is Occam[5]. Occam is a parallel programming language developed based on communication sequential processes[11]. To execute Occam, a hardware is designed by Inmos in the form of very large scale integration(VLSI) integrated chip(IC), which is called the Transputer. Occam and the Transputer are mainly intended to be built into hand-held gadgets. Hence, simplicity and security are the main concerns of the design. Occam supports concurrency on both single and multiple transputers. But they are achieved only by means of messages passing along channels. The communication on an Occam channel is synchronous. Whether a sender or a receiver arrives at the synchronisation point, the first is to wait for the other. Once they are synchronised, the message is transferred between the two. For simplicity, concurrency in Occam can be expressed explicitly at the statement level with PAR construct. Occam does not support pointers, dynamic memory or process allocation, which makes it easier to debug and safer.

Synchronous communication is not suitable for multimedia processing, which usually involves high rate data flow coming from all the nodes. The cost of waiting for the synchronisation point in a synchronous communication scheme is obviously too high for multimedia processing.

4.4 Conclusions

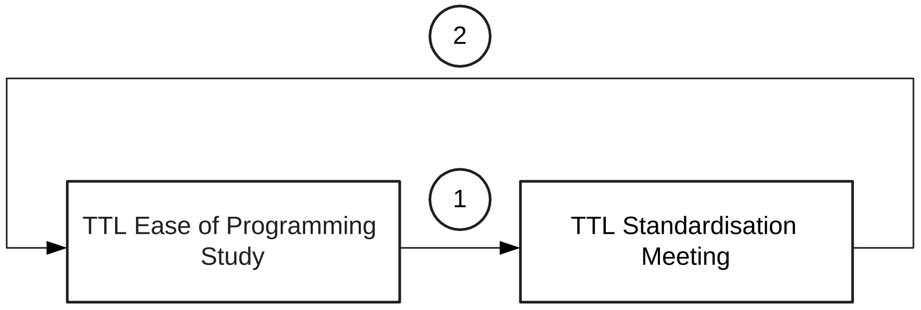
From the study of the related work, we have learnt that in data communication domain, circuit switching is more suitable for multi-media processing applications than packet switching; in embedded system domain, shared-memory communication is more efficient than message passing communication in shared memory address space architectures. We have also shown four realtime infrastructure examples from other teams, which give us more ideas of inter-task communication implementation. Having the knowledge, we will develop a methodology to assess the design effort required from different TTL design choices in the next chapter.

Methodology 5

The TTL interface design choices need to be evaluated in the aspect of ease of programming, as was explained in Chapter 3. How can we achieve that? In this chapter, we discuss the methodology that we use to assess the design effort each proposed interface requires and assist the TTL interface standardisation. In Section 5.1, the role of the TTL ease of programming study project in Philips Research is explained. In Section 5.2, the solution approach is described. This chapter is concluded in Section 5.3.

5.1 Project Role

This project is part of the Task Transaction Level (TTL) Interface project in ESAS Group, IST Sector, Philips Research. The result of the TTL ease of programming study project serves as an input to the TTL standardisation group meetings where TTL Interface design proposals are discussed. The open issues or new decision proposals from the meeting which concerns application ease of programming are further studied in the TTL ease of programming study project. Here the procedure is visualised in Figure 5.1:



Data Flow:
1: Conclusions and code examples on design decision points in ease of programming aspect
2: Design proposals

Figure 5.1: The role of TTL ease of programming study in TTL interface design early phase

5.2 Approach

The idea of using benchmark applications to realise quantitative performance analysis can also be used to evaluate design effort. The approach involves the following steps:

1. Summarise and categorise communication behaviours in media processing applications;
2. Select typical multimedia processing code examples for different communication behaviour classes;
3. Define ease of programming criteria and weight factors;
4. Study the proposed TTL design choices;
5. Define a comparison scheme;
6. Rank the proposed TTL interface according to the criteria, weight factors and the selected code examples;
7. Compare and conclude the TTL design choices in the aspect of ease of programming.

The definition of a comparison scheme involves the selection of scenarios for each set of design choices. Here we define a scenario to be a group of code examples applying an evaluated design choice.

The rank of a scenario can be obtained with the following formula:

$$S = \sum(w_i \times r_i)$$

where S stands for the final score, w_i stands for the weight factor and r_i is the rate for the criterion.

The specific definitions of the weight factors will be discussed in the next chapter. A higher rate for a criterion means less design effort required. The higher full score a scenario has, the easier it is to be programmed.

Comparison of the design choices thus can be done by comparing the full scores calculated for certain scenarios selected according to the comparison scheme.

A simple example of evaluation can be given for the two TTL applications in Chapter 3. Assume the criteria are defined as:

Criteria	**Weight Factors**
1. Complexity of data structure	2
2. Lines of code	1

Table 5.1: A simple example of criteria and weight factors

We rank each of the applications as follows:

From the evaluation, we conclude that with the design choice used in TTL1 it is easier to program than with the one used in TTL2.

Following the above described steps, we will have a quantitative evaluation for the design choices in the aspect of ease of programming in the application designers' point of view.

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Complexity of data structure	Line 5, 6, 7, 8, 10, 11: Only simple data structures are needed.	2	3
2. Lines of code	Line 5, 6, 7, 8, 10, 11: Three communication primitives are needed	1	2
Final Score:			8

Table 5.2: Evaluation of TTL1

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Complexity of data structure	Line 3, 4, 11, 12: Array data structure is needed.	2	2
2. Lines of code	Line 5, 6, 7, 8, 10, 11: Three communication primitives are needed	1	2
Final Score:			6

Table 5.3: Evaluation of TTL2

5.3 Conclusions

In this chapter, we explained the role of the TTL ease of programming study project and developed a seven-step approach for the project. Each step will be elaborated in the coming chapters.

6 Code Examples and Evaluation Criteria

As described in the last chapter, the evaluation process consists of seven steps. This chapter will discuss the first four steps. In Section 6.1, the investigation of communication behaviours in media processing applications is discussed; in Section 6.2, we explain the second step, namely the selection of the benchmark code examples; the definition of the criteria and weight factors is described in Section 6.3. Finally we conclude the chapter in Section 6.4.

6.1 Characteristics of Inter-Task Communication in Multimedia Processing

In TTL, tasks exchange streamed data via channels. Sending or receiving data to a channel is commonly implemented in two steps: synchronisation and data transfer.

Synchronisation is a step in which the initiated task or the infrastructure makes sure that the communication partner is ready to communicate. In TTL, it further indicates that there are required number of tokens available in the channel for the task.

Data Transfer copies the value from a token in a channel to a local variable or the value of a local variable to a token in a channel.

Here we define a token to be a unit of data transfer. If high level communication data types (We will explain that in more details in next chapter.) are used, a token can be of any system or user defined type that is allowed. If low level communication data types are used, a token is a bit or a bit vector.

Communication behaviours in typical multimedia processing applications can be categorised into two classes in the following two ways:

- Data-dependent behaviour and Data-independent behaviour: Data-dependent behaviour indicates the number of tokens to be transferred is unknown until run time. However, the number is already defined during compile time for data-independent communication processes.

- Deterministic behaviour and Non-deterministic behaviour: Deterministic processes will produce the same result for given input, independent of the scheduling of the processes. For non-deterministic processes, for instance, interactive applications, the arrival time of the user input is non-deterministic and will affect the result of the execution.

6.2 Benchmark Code

To study the different design efforts that are needed when different TTL primitives are chosen, we select four YAPI processes: *idct1d*(1 dimensional inverse DCT), *hs* (horizontal scalar), *izz*(inverse zigzag) and *filter*(TV brightness control) as illustrative application code.

idct1d and *izz* are typical examples of deterministic, data-independent communication behaviour applications. *hs* is an example of deterministic communication and data-dependent communication behaviour application. And *filter* is an example for the non-deterministic data-independent communication behaviour application.

idct1d, *hs* and *izz* are all from a jpeg decompression application. And *filter* can be used as a process to realise TV brightness control. The functionality of the four processes are explained as below:

- *idct1d*: one dimensional inverse DCT
  - input: 8 pixels in frequency domain
  - output: 8 pixels in time domain
- *hs*: horizontal scalar, which together with vertical scalar subsample the components (usually Y, Cb, Cr) such that all the components have the size of the image. The output of hs is the input of matrix, a process that converts the Y, Cb, Cr values to R, G, B values for each pixel.
  - input: pixels
  - input: line width of input pixels
  - input: line width of output pixels
  - output: pixels
- *izz*: inverse zigzag
  - input: 8×8 zigzagged pixels in frequency domain
  - output: 8×8 pixels in left-to-right, top-to-bottom sequence in frequency domain
- *filter*: TV brightness control
  - input: real-time brightness enlargement coefficient user input
  - input: pixels
  - output: pixels

The YAPI implementation fragments of the four processes are presented as below. Communication functions and their annotation are highlighted.

CASE 1: idct1d
The inline function definitions for ADD, SUB, CMUL and but, which are not TTL related are skipped here.

```
/* idct algorithm */ void idct(VYApixel *Y)
{
    VYApixel z1[8], z2[8], z3[8];

    // Stage 1:
    but(Y[0], Y[4], z1[1], z1[0]);
    z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
    z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
    but(Y[1], Y[7], z1[4], z1[7]);
    z1[5] = CMUL(23170, Y[3]);
    z1[6] = CMUL(23170, Y[5]);

    // Stage 2:
    but(z1[0], z1[3], z2[3], z2[0]);
    but(z1[1], z1[2], z2[2], z2[1]);
    but(z1[4], z1[6], z2[6], z2[4]);
    but(z1[7], z1[5], z2[5], z2[7]);

    // Stage 3:
    z3[0] = z2[0];
    z3[1] = z2[1];
    z3[2] = z2[2];
    z3[3] = z2[3];
    z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
    z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
    z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
    z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

    // Stage 4:
    but(z3[0], z3[7], Y[7], Y[0]);
    but(z3[1], z3[6], Y[6], Y[1]);
    but(z3[2], z3[5], Y[5], Y[2]);
    but(z3[3], z3[4], Y[4], Y[3]);

}

/* Inverse 1-D Discrete Cosine Transform.
   Result Y is scaled up by factor sqrt(8).
   Original Loeffler algorithm.
*/ void IDCT1D::main()
{
        VYApixel Y[8];
        VYApixel z1[8], z2[8], z3[8];

        while (true)
        {
            // read a vector of 8 values from input port CIn to array Y.
             read(CIn, Y, 8);
            idct(Y);
            // write a vector of 8 values from array Y to output port COut.
             write(COut, Y, 8);
        }
}
```

CASE 2: hs

```
void HS::main()
{
    while (true)
    {
        VYApixel        pixel;
        VYAlineLength   inLineLength;
        VYAlineLength   outLineLength;
        unsigned int    scaleFactor;

        read(inLineLengthInP, inLineLength);
        read(outLineLengthInP, outLineLength);

        scaleFactor = ceil_div(outLineLength,inLineLength);

        /* The number of pixels to read and write is dependent on the
           input of inLineLengh and outLineLength, which is unknown
           until runtime. */
        for (unsigned int i=0; i<inLineLength; i++)
        {
            read(CinP, pixel);
            unsigned int m = i*scaleFactor;
            unsigned int n = std::min(m+scaleFactor,outLineLength);
            for (unsigned int j=m; j<n; j++)
            {
                write(CoutP, pixel);
            }
        }
    }
}
```

CASE 3: izz

```
void IZZ::main()
{
    VYApixel    Cin[64];
    VYApixel    Cout[64];

    while (true)
    {
        read(CinP, Cin, 64);
        for (unsigned int i=0; i<64; i++)
        {
            Cout[zigzag[i]] = Cin[i];
        }
        write(CoutP, Cout, 64);
    }
}
```

CASE 4: filter

```
void Filter::main()
{
    int j, k;
    int coeff = 1;
    int* v = new int [npixels];

    while (true)
    {
        /* The input time in channel cof is undeterministic.
           Once it comes, the value of coeff needs to be updated.
        */
        if ( select(in, cof) == 1)
        {
             read(cof, coeff);
        }

        for (j=0; j<nlines; j++)
        {
            /* npixels is a predefined global value, which is known
              at compile time. */
             read(in, v, npixels);
            for (k=0; k<npixels; k++)
            {
                 write(out, coeff*v(k));
            }
        }
    }
    delete [] v;
}
```

6.3 Criteria and Weight factors

In order to describe the level of ease of programming in a quantitative way, we define the criteria and weight factors as presented in the table below:

Criteria	Weight Factors
1. Number of communication primitives needed	1
2. Number of parameters needed in communication primitives	1
3. Lines of code	1
4. Number of loops	1
5. Complexity of data structures	2
6. Code readability	3
7. Number of applicable cases	1
8. Error-proneness	3
9. Code reusability	3
10. Memory usage	3

Table 6.1: criteria and weight factors

The weight factors are defined according to an ordinal scale. The weight factor corresponding to each criterion indicates how important the criterion is in comparison with others. For example, a criterion with weight 3 is considered to be more important than a criterion with weight 1. The weight factors are defined in the scale of 1 to 3.

The reason to consider memory usage as a criterion for ease of programming is because the amount of memory available in the target embedded system is often crucial and often has impact on the algorithms of the applications. The fact that some proposed TTL interfaces can alleviate the demand to system memory can widen the choices for algorithms, thus will also relieve the ease of programming.

6.4 Conclusions

In this chapter, how we performed the first four steps of the approach defined in the last chapter were described. We showed that inter-task communication behaviours in multimedia processing can be categorised into two classes in two ways: data-dependent behaviour and data-independent behaviour; deterministic behaviour and non-deterministic behaviour. According to that, a set of benchmark code is selected. Finally, we proposed ten criteria and weight factors for the quantitative ease of programming evaluation.

7 Design Choices and Comparison Scheme

This chapter is focused on the study of TTL design choices on inter-task communication and the scheme we will use in order to compare the design choices. TTL inter-task communication is how communication among tasks takes place in TTL. The design choices in this topic are about which inter-task communication services TTL provides and which not. This is discussed in Section 7.1. In Section 7.2, a comparison scheme is proposed. In this comparison scheme, we define a set of scenarios for the benchmark code examples according to the design choices and indicate the comparison of which scenarios can conclude each design choice.

7.1 TTL Inter-task Communication Design choices

To define the specification for TTL inter-task communication, we have a list of about twenty issues that will affect the TTL primitives on inter-task communication. The list is presented in Appendix B. Among those issues, some already have confirmed answers out of the consideration of system architecture before the student project started, and the rest are still under discussion. In this document, we will study the uncertain issues and compare them in the aspect of ease of programming.

The TTL decision points we decide to study are:[10]

1. **Combined vs. Separate Synchronisation and Data Transfer**

   Combined Synchronisation and Data Transfer:
   With Combined Synchronisation and Data Transfer, sending or receiving a token from one process to another can be done with only one primitive: `write(c, ...)` or `read(c, ...)` respectively.

   Separate Synchronisation and Data Transfer:
   In a Separate Synchronisation and Data Transfer scheme, sending or receiving a data is split into three operations:

   Producer's side:

   `acquireRoom(c, ...)`:Acquire (an) empty token(s) from channel c.

   `store(c, v, ...)`: Transfer (a) value(s) from the producer task into the acquired empty token.

   `releaseData(c, ...)`: Release the filled token(s) to the channel.

   Consumer's side:

   `acquireData(c, ...)`: Acquire (a) full token(s) from channel c.

`load(c, v, ...)`: Transfer (a) value(s) from the full token into the consumer task.

`releaseRoom(c, ...)`: Release (an) emptied token(s)into the channel.

The main reasons of proposing this decision point to study are because combined synchronisation and data transfer needs less primitives, independent of the buffer size of the channel, however separate synchronisation and data transfer improves efficiency by reducing bus transfers when local memory is not sufficient for private variables.

2. **Scalar vs. Vector Operations**

Scalar Data Operation:
With a Scalar Data Operation, only one token is processed in one operation.

Applied this to the cases of combined synchronisation and data transfer, the primitives will be like the following:

- Combined, Scalar Synchronisation and Data Transfer:
  `write(c, v)`: Write a value v to the channel c.
  `read(c, v)`: Read a value v from the channel c.

When there are more tokens to be transferred, we can define a data type which contains a fixed size of array and use the new data type as a token. We call it Scalar with Array Data Type.

- Combined Scalar with Array Data Type Synchronisation and Data Transfer
  The primitives for Combined Scalar with Array Data Type Synchronisation and Data Transfer are the same as those for Combined, Scalar Synchronisation and Data Transfer.

When scalar operation is used with separate synchronisation and data transfer, there are more issues to be discussed.

Vector Data Transfer:
A sequence of tokens can be also handled once by introducing a vector operation. The effect is that all the primitives for vector data transfer will have one more parameter n to indicate the number of elements in the vector v.

- Combined, Vector Synchronisation and Data Transfer:
  `write(c, v, n)`: Write a vector v of n elements to the channel c.
  `read(c, v, n)`: Read a vector v of n elements from the channel c.

The design choice is proposed to study because in some systems, scalar operations can be implemented more efficiently. However in other systems, vector operations can be implemented more efficiently.

The comparison of Combined Scalar Synchronisation and Data Transfer, Combined Scalar with Array Data Type Synchronisation and Data Transfer and Combined

Vector Synchronisation and Data Transfer can show us the difference between scalar operation and vector operation.

Vector operation used with separate synchronisation and data transfer will lead to more issues to be discussed.

3. **In-Order vs. Out-of-Order Intra-task Synchronisation**

   In-Order Intra-task Synchronisation:
   Within a task, a producer or a consumer, if a token is released in the same order as the token is acquired before, we call it in-order intra-task synchronisation. This only applies to the situation when separate synchronisation is used.

   Out-of-Order Intra-task Synchronisation:
   Within a task, a producer or a consumer, if a token or a room is released in a different order than the room or the token is acquired before, we call it out-of-order intra-task synchronisation. Like in-order intra-task synchronisation, this only applies to the situation when separate synchronisation is used.

   The reason to propose the design choice to study is because in-order intra-task synchronisation can be implemented more efficiently in the sense of execution time when memory utilisation is not an issue, however out-order intra-task may reduce the memory requirements by releasing a token once it is not needed anymore, no matter the earlier acquired tokens are released or not.

4. **Direct vs. Indirect Access**

   Direct Access:
   In the direct access scheme, data transfer is done by accessing the token directly with the address to a memory location which is obtained from the acquire function. Parts of a big token can be addressed by adding offsets to the start address of the token.

   - Separate, Vector, In-Order Intra-task Synchronisation and Direct Access
     Producer's side:

     `acquireRoom(c, o, n)`: Acquire n empty tokens from channel c, and store the identifiers in the vector o of length n.

     `releaseData(c, n)`: Release the n oldest acquired tokens.

     Consumer's side:

     `acquireData(c, i, n)`: Acquire n full tokens from channel c, and store the identifiers in the vector i of length n.

     `releaseRoom(c, n)`: Release the n oldest acquired tokens.

   - Separate, Vector, Out-of-Order Intra-task Synchronisation and Direct Access
     Producer's side:

     `acquireRoom(c, o, n)`: Acquire n empty tokens from channel c, and store the identifiers in the vector o of length n.

`releaseData(c, o, n)`: Release the tokens identified by the vector o of length n.

Consumer's side:

`acquireData(c, i, n)`: Acquire n full tokens from channel c, and store the identifiers in the vector i of length n.

`releaseRoom(c, i, n)`: Release the tokens identified by the vector i of length n.

The scalar version of the primitives leaves the parameter n out.

Comparison of the above two cases: Separate, Vector, In-Order Intra-task Synchronisation and Direct Access and Separate, Vector, Out-of-Order Intra-task Synchronisation and Direct Access will show us the advantages and disadvantages of in-order intra-task synchronisation and out-of-order intra-task synchronisation.

Indirect Access:
In the indirect access scheme, data transfer must be done explicitly via functions, e.g. load and store, which acquire the channel information and token reference identifiers. A token reference identifier can be a pointer to a memory location, an index in an array, or other data types.

The decision point comes from the fact that direct access is more efficient for large tokens and indirect for small tokens.

5. **Absolute vs. Relative Token References**

Absolute Token References:
This only applies to indirect access. An absolute token reference is to address a token with a reference which is obtained from the acquire function. And the reference will be used in data transfer function, e.g. load/store.

- Separate, Vector, In-Order Intra-task Synchronisation and Absolute References
  Producer's side:

  `acquireRoom(c, o, n)`: Acquire n empty tokens from channel c and store the identifiers in the vector o of length n.

  `store(c, o, v, n)`: Copy n values from local vector v of length n to the acquired tokens referred by identifier vector o.

  `releaseData(c, n)`: Release the oldest n tokens.

  Consumer's side:

  `acquireData(c, i, n)`: Acquire the next n full tokens from channel c and store the identifiers in the vector i of length n.

  `load(c, i, v, n)`: Copy n values of tokens referred by identifier vector i to local vector v of length n.

  `releaseRoom(c, n)`: Release the oldest n tokens.

  When it is used in scalar operation, the parameter n is left out.

- Separate, Vector, Out-of-Order Intra-task Synchronisation and Absolute References

  Producer's side:

  `acquireRoom(c, o, n)`: Acquire n empty tokens from channel c and store the identifiers in the vector o of length n.

  `store(c, o, v, n)`: Copy n values from local vector v of length n to the acquired tokens referred by identifier vector o.

  `releaseData(c, o, n)`: Release the n tokens addressed by the vector of o.

  Consumer's side:

  `acquireData(c, i, n)`: Acquire the next n full tokens from channel c and store the identifiers in the vector i of length n.

  `load(c, i, v, n)`: Copy n values of tokens referred by identifier vector i to local vector v of length n.

  `releaseRoom(c, n)`: Release the n tokens addressed by the vector of i.

  When it is used in scalar operation, the parameter n is left out.

To compare the differences between Separate, Vector, In-Order Intra-task Synchronisation and Direct Access and Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Absolute References can show us the differences between direct access and indirect access.

Relative Token References:
When in-order intra-task synchronisation and indirect access is used, we can address a token with an integer value which indicates the relative distance to the oldest token that has been acquired but not yet been released. We call this relative token reference. When we apply them to vector operation, the primitives for that are:

- Separate, Vector, In-Order Intra-task Synchronisation and Relative References

  Producer's side:

  `acquireRoom(c, n)`: Acquire n empty tokens from channel c.

  `store(c, d, v, n)`: Copy a vector v of n values into the tokens with distance d, , d + n - 1 to the oldest acquired token in channel c.

  `releaseData(c, n)`: Release the n oldest acquired tokens to channel c.

  Consumer's side:

  `acquireData(c, n)`: Acquire n full tokens from channel c.

  `load(c, d, v, n)`: Copy the value of the tokens with distance d, , d + n - 1 to the oldest acquired token into vector v of length n.

  `releaseRoom(c, n)`: Release the n oldest acquired tokens to channel c.

  When it is used in scalar operation, the parameter n is left out.

Absolute reference makes it possible to transfer only part of a large token but for small tokens it has too much overhead to obtain and maintain absolute references for all the tokens. Relative reference does not require a task to hold the absolute references but requires a complex administration to keep track on which tokens have been acquired and not yet released when it is combined with out-of-order intra-task communication.

The difference of relative token references and absolute token references can be seen by comparing the above two cases: Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Relative References and Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Absolute References.

6. **Blocking vs. Non-blocking Synchronisation**

   Blocking Synchronisation
   A blocking synchronisation operation will not return until the communication partner has reached the synchronisation point. All the primitives discussed above belong to blocking synchronisation.

   Non-blocking Synchronisation:
   A non-blocking synchronisation operation will return immediately. The return value depends on whether the communication peer has reached the synchronisation point or not.

   The trade-off between blocking and non-blocking synchronisation is that blocking synchronisation hides the postponement implementation from the application designers thus easier to use but non-blocking synchronisation can reduce the number of context switches or avoid busy waiting.

7. **Interrupt Service Routines vs. Busy Waiting Polling**

   Interrupt Service Routines:
   If task switch is supported by a system, applications using Non-blocking Synchronisation can be implemented as an Interrupt Service Routine. A task can return if a synchronisation operation is not successful. The system will take over and schedule to another task. Once there is any change in a channel, the returned task will be called again.
   Busy Waiting Polling:
   When Non-blocking synchronisation is used, the synchronisation step can also be implemented by a application designer with busy waiting polling. With busy waiting, the task uses a while loop to repeatedly check if the required tokens or rooms are ready.

   When it is not too costly for a system to do context switch and the operating system supports interrupt service routine, implementing tasks in the way of interrupt service routine may bring higher performance to a system. But some systems do not support task switch due to the cost, busy waiting is a solution for non-blocking synchronisation.

8. **Test and Re-Acquire vs. Test and Acquire**

   Test and Re-Acquire:
   A non-blocking synchronisation can be done with a test and re-acquire operation or a test and acquire operation. With a test and re-acquire operation, an acquire operation will test the availability of the defined number of tokens in the channel. The checking starts with the first token in the channel that has not yet been released. So sequential acquire operations for the same number of tokens on the same channel before a release operation will test the availability of the same tokens. Applying this to vector operation, the primitives are:

   - Non-blocking, Vector Synchronisation, Test-only

     `bool testReAcquireRoom(c, n):` Return true if at least n empty tokens are available on channel c, and return false otherwise.

     `bool testReAcquireData(c, n):` Return true if at least n full tokens are available on channel c, and return false otherwise.

     The scalar version of the primitives is without the parameter n.

   We compare Non-blocking, Vector Synchronisation, Test and Re-Acquire used in Interrupt Service Routines with Separate, Vector, In-Order Intra-task Synchronisation and Relative References to see the advantages and disadvantages of non-blocking synchronisation and blocking synchronisation.

   By comparing Non-blocking, Vector Synchronisation, Test and Re-Acquire used in Interrupt Service Routines with Non-blocking, Vector Synchronisation, Test and Re-Acquire with Busy Waiting, we can draw conclusion on the difference between interrupt service routines method and busy waiting polling method.

   Test and Acquire:
   A test and acquire operation will return after the tokens are acquired. A sequential acquire operation will check the next tokens in the channel. The primitives for vector operation are:

   - Non-blocking, Vector Synchronisation, Test and Acquire

     `bool testAcquireRoom(c, n):` Acquire n empty tokens and return true if at least n empty tokens are available on channel c, and return false otherwise.

     `bool testAcquireData(c, n):` Acquire n full tokens and return true if at least n full tokens are available on channel c, and return false otherwise.

     The scalar version of the primitives is without the parameter n.

   Test and Reacquire is useful when it is impossible or expensive to save and store the state of a task. Test and Acquire is useful when reloading a token from a channel is more expensive than saving and restoring the state.

9. **High-level vs. Low-level Communication Data Types**

   High-level Communication Data Types
   If high-level communication data types are supported, the TTL infrastructure knows the semantics of data types and hides the endianness issues from the tasks.

   Low-level Communication Data Types:
   With this type of service, the tasks can only send data in the form of (multiple) bits. Issues like semantics of data types and endianness are left to the tasks to deal with.

   To compare Low-level Communication Data Types, Combined, Vector, Synchronisation and Data Transfer with Combined, Vector, Synchronisation and Data Transfer can show us the difference in using high-level communication data types and low-level communication data types.

   High-level Communication Data Types make tasks more reusable and easy to design but low-level Communication Data Types are needed when communication of different data types is done via a single-type channel.

   The primitives discussed until now are sufficient to realise the synchronisation and data transfer in the case of deterministic communication behaviour, either data-dependent or data-independent. However synchronisation of non-deterministic behaviour is not covered yet.

Synchronisation of non-deterministic communication
To achieve the synchronisation of the non-deterministic processes, TTL is suggested to support a primitive select corresponding to the YAPI primitive with the same name. The syntax of the primitive is:

- Scalar Synchronisation:
  `r = select(port1, port ,   , portn)`
  The function tests the availability of a token (room) in each channel (port1 to portn) and will fill the status of each channel into the bit vector r and return until there is at least one channel is ready for communication. (Bit 1 for ready and bit 0 for unready)

- Vector Synchronisation
  `r = select(port1, n1, port2, n2,  portn , nn)`
  The function tests the availability of defined number (n1 for port1, , nn for portn) of tokens (rooms) in each channel (port1 to portn) and will fill the status of each channel into the bit vector r and return until there is at least one channel is ready for communication. (Bit 1 for ready and bit 0 for unready)

The two `selcet()` primitives have blocking behaviour and primitives testReAcquireData and testReAcquireRoom have the same effect on non-deterministic communication except for its non-blocking characteristic.

7.2 Comparison Strategy

To study the different design efforts that are needed when different TTL primitives are chosen, we select four YAPI processes: idct1d (1 dimensional inverse DCT), hs (horizontal scalar), izz (inverse zigzag) and filter(TV brightness control) as illustrative application code, as we explained in Section 3.2. For each YAPI process, we will create the following versions (if applicable) of TTL applications (except for filter, we will skip some of the versions which apparently only have effects on the deterministic communication part):

1. Combined Scalar Synchronisation and Data Transfer
2. Combined Scalar with Array Data Type Synchronisation and Data Transfer
3. Combined Vector Synchronisation and Data Transfer
4. Separate, Vector, In-Order Intra-task Synchronisation and Direct Access
5. Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Direct Access
6. Separate, Out-of-Order Intra-task Synchronisation and Direct Access
7. Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Absolute References
8. Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Relative References
9. Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Indirect Access with Relative References
10. Non-blocking, Vector Synchronisation, Test and Re-Acquire used in Interrupt Service Routines with Relative References
11. Non-blocking, Vector Synchronisation, Test and Re-Acquire with Busy Waiting and Relative References
12. Non-blocking, Vector Synchronisation, Test and Acquire used in Interrupt Service Routines with Relative References
13. Non-blocking, Vector Synchronisation, Test and Acquire with Busy Waiting and Relative References
14. Low-level Communication Data Types, Combined, Vector, Synchronisation and Data Transfer

Note that items 1-9 are blocking synchronisation and items 1-14 are all high-level communication data types.

After the comparison of the different TTL translations, we can see which primitive choices are better in the sense of ease of programming. The comparison strategy is as listed below:

Decision Points	**Compared TTL Versions**
1. Combined vs. Separate	3 vs. 4
2. Scalar vs. Vector	1 vs. 2 vs. 3, 4 vs. 5, 8 vs. 9
3. In-Order vs. Out-Of-Order	4 vs. 6
4. Direct vs. Indirect	4 vs. 7
5. Absolute vs. Relative	7 vs. 8
6. Blocking vs. Non-blocking	8 vs. 10
7. Interrupt Service Routines vs. Busy Waiting Polling	10 vs. 11, 12 vs. 13
8. Test and Re-Acquire vs. Test and Acquire	10 vs. 12
9. High-level vs. Low-level	3 vs. 14

Table 7.1: Comparison Strategy

Combined Synchronisation and Data Transfer 8

Following the steps described in Chapter 5, we are now up to the sixth step: rank the proposed TTL interfaces. In this chapter, the ease of programming of combined synchronisation and data transfer is studied. As explained in the last chapter, combined synchronisation and data transfer indicates sending or receiving one or more tokens with only one primitive. The main purpose of this chapter is to make comparison among scalar, scalar with array data type and vector operations. They are discussed in three sections respectively.

This chapter and the later chapters are organised in the following way: For each scenario, we give a brief description of the design choice and then explain how the design choice is applied to the code examples, the adapted code is presented afterwards and in the end we motivate and conclude the evaluation.

8.1 Combined Scalar Synchronisation and Data Transfer

Description: With combined scalar synchronisation and data transfer, tokens are sent or received one by one with the primitives: read(c, v) or write(c, v), where c is the channel and v is the value to be transferred.

Suggested Solution: Use a loop with a scalar operation instead of a vector operation in the original code.

CASE 1: idct1d

```
void idct(VYApixel *Y)
{
   ...
}

/* Inverse 1-D Discrete Cosine Transform.
   Result Y is scaled up by factor sqrt(8).
   Original Loeffler algorithm.
*/
void IDCT1D::main()
{
      VYApixel Y[8];
      VYApixel z1[8], z2[8], z3[8];

      while (true)
      {
            /* An extra loop is needed to change
```

```
       it from the original vector
       operation. */
     for ( int i=0; i¡8; i++ )
     {
       /* For each data transfer, only one
          token space is required
          from local memory. */
        read(CIn, Y[i]);
     }

        idct(Y);

         for ( int i=0; i¡8; i++)
     {
         write(COut, Y[i]);
     }
   }
}
```

CASE 2: hs

```
void HS::main()
{
  while (true)
  {
      VYApixel        pixel;
      VYAlineLength   inLineLength;
      VYAlineLength   outLineLength;
      unsigned int        scaleFactor;

       read(inLineLengthInP, inLineLength);
    read(outLineLengthInP, outLineLength);

      scaleFactor = ceil_div(outLineLength,inLineLength);

      for (unsigned int i=0; i<inLineLength; i++)
      {
           read(CinP, pixel);
          unsigned int m = i*scaleFactor;
          unsigned int n = std::min(m+scaleFactor,outLineLength);
          for (unsigned int j=m; j<n; j++)
          {
              write(CoutP, pixel);
          }
      }
  }
}
```

CASE 3: izz

```
void IZZ::main()
{
  VYApixel    Cin[64];
VYApixel    Cout[64];

while (true)
{
     for (unsigned int i=0; i¡64; i++)
    {
         read(CinP, Cin[i]);
        Cout[zigzag[i]] = Cin[i];
     }

     for (int i=0; i¡64; i++)
    {
      write(CoutP, Cout[i]);
    }
 }
}
```

Evaluation: From the above three cases, we can see combined scalar synchronisation and data transfer

+ is very easy to use when there is only one variable that is needed to transfer;

– but seems tedious when multiple tokens are to be handled at one time, however sometimes this problem can be solved by using scalar with array data type which we will discuss in the following section.

Summary: With the criteria we defined in an early chapter, we can rank combined scalar synchronisation and data transfer as in Table 8.1:

8.2 Combined Scalar with Array Data Type Synchronisation and Data Transfer

Description: Instead of sending and receiving small tokens one by one, we can define a data type that contains a fixed number of single tokens such that we can use one scalar operation to transfer a block of small tokens. This approach is called scalar with array data type.

Suggested Solution: In the case of data-independent behaviour, the number of tokens to be transferred is known at compile time. We can thus define a structure with an array which holds the number of tokens. If it is data-dependent communication, the number of tokens to be transferred remains unknown until run time. We will then have to just define a structure with a reasonable size and possibly transfer multiple blocks according to the actual token numbers.

CASE 1: idct1d

This is a case of data-independent communication.

```
/* Define a structure with the number tokens
  needed to be transferred at one time. */
```

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 12, 17; hs line 9, 10, 14, 19; izz line 9, 14: To send or receive a token, only one argument is needed.	1	3
2. Number of parameters needed in communication primitives	idct1d line 12, 17; hs line 9, 10, 14, 19; izz line 9, 14: Each of read and write needs only two parameters.	1	3
3. Lines of code	idct1d line 10, 15; izz line 7, 12: Extra loops increase the number of the lines.	1	1
4. Number of loops	idct1d line 10, 15; izz line 7, 12: Extra loops are added to realise multi-token operations.	1	2
5. Complexity of data structures	idct1d line 12, 17; hs line 9, 10, 14, 19; izz line 9, 14: Original token types are used for communication.	2	3
6. Code readability	idct1d line 10, 12, 15, 17; izz line 7, 9, 12, 14: To use scalar operation inside loops makes code less readable than to use vector operation.	3	2
7. Number of applicable cases	It is suitable for any communication behaviour type: data dependent or independent.	1	3
8. Error-proneness	No pointers and difficult data structures are used. Change of one part of code does not affect the other part.	3	3
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 12, 17; hs line 9, 10, 14, 19; izz line 9, 14: For each communication session, only local memory size of a single token is required.	3	3
Final Score:			50

Table 8.1: Evaluation of Combined Scalar Synchronisation and Data Transfer

```
1 struct VYABlock˙type
2 {
3  VYApixel Y[8];
4 };

/* Inverse 1-D Discrete Cosine Transform.
   Result Y is scaled up by factor sqrt(8).
   Original Loeffler algorithm.
*/
5 void IDCT1D::main()
6 {
7       VYApixel z1[8], z2[8], z3[8];
8       struct VYABlock˙type VYABlock;


9       while (true)
10      {
```

```
            /* A big block of local memory is needed. */
            read(CIn, VYABlock);

           // Stage 1:
           but(VYABlock.Y[0], VYABlock.Y[4], z1[1], z1[0]);
           z1[2] = SUB(CMUL( 8867, VYABlock.Y[2]), CMUL(21407, VYABlock.Y[6]));
           z1[3] = ADD(CMUL(21407, VYABlock.Y[2]), CMUL( 8867, VYABlock.Y[6]));
           but(VYABlock.Y[1], VYABlock.Y[7], z1[4], z1[7]);
           z1[5] = CMUL(23170, VYABlock.Y[3]);
           z1[6] = CMUL(23170, VYABlock.Y[5]);

           // Stage 2:
           but(z1[0], z1[3], z2[3], z2[0]);
           but(z1[1], z1[2], z2[2], z2[1]);
           but(z1[4], z1[6], z2[6], z2[4]);
           but(z1[7], z1[5], z2[5], z2[7]);

           // Stage 3:
           z3[0] = z2[0];
           z3[1] = z2[1];
           z3[2] = z2[2];
           z3[3] = z2[3];
           z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
           z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
           z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
           z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

           // Stage 4:
           but(z3[0], z3[7], VYABlock.Y[7], VYABlock.Y[0]);
           but(z3[1], z3[6], VYABlock.Y[6], VYABlock.Y[1]);
           but(z3[2], z3[5], VYABlock.Y[5], VYABlock.Y[2]);
           but(z3[3], z3[4], VYABlock.Y[4], VYABlock.Y[3]);

            write(COut, VYABlock);
  }
}
```

CASE 2: hs

This example contains both data-independent behaviour and data-dependent behaviour.

```
/* Here the size of the structure is not the number of tokens to be transferred. The actual number of tokens can be bigger or smaller than the number defined. */
struct VYABlock˙type
{
  VYApixel Y[8];
};

#define VYABLK˙LEN 8

void HS::main()
{
  unsigned int blklen = VYABLK˙LEN;
  /* offset indicates the start processing point in a inLineBlk. */
  unsigned int offset=0;
```

```
while (true)
{
    VYApixel            pixel;
    VYAlineLength       inLineLength;
    VYAlineLength       outLineLength;
    unsigned int            scaleFactor;
    struct VYABlock˙type    inLineBlk, outLineBlk;
   /* h is the writing pointer in a outLineBlk. */
    unsigned int           h = 0;

   read(inLineLengthInP, inLineLength);
   read(outLineLengthInP, outLineLength);

    scaleFactor = ceil_div(outLineLength,inLineLength);

  /* i is the index of the pixels in a line. */
   unsigned int i=0;
   while ( i¡inLineLength )
   {
     /* If last inLineBlk contains the end part of
       last line and the beginning part of the
       current line, offset != 0.
      */
      if (offset==0)
      {
         read(CinP, inLineBlk);
      }

      unsigned int len = std::min(blklen, inLineLength-i);
     /* k is the index of the pixels in a blk.*/
      for(unsigned int k=offset; k¡len; k++)
     {
            unsigned int m = (i+k)*scaleFactor;
            unsigned int n = std::min(m+scaleFactor,outLineLength);
            for (unsigned int j=m; j<n; j++)
            {
                outLineBlk.Y[h] = inLineBlk.Y[k];
                h++;
                if (h == blklen)
                {
                   write(CoutP, outLineBlk);
                   h = 0;
                }
            }

        /* Calculate from which pixel in a blk the next loop
          should start to process.
         */
         offset = len % blklen;
     }

     /* Calculate from which pixel in the line the next loop should
       start to process.
      */
      i += blklen-offset;
```

```
        }
        if (h!=0)
        {
            write(CoutP, outLineBlk);
        }
    }
}
```

For the above change, the task matrix which receives input from hs needs an extra input for inLineLength so that it knows how many pixel values it should extract from the input blocks. This can be achieved by adding a FIFO between task vs(vertical scalar) and task matrix.

The example above shows that scalar with array data type is rather complex in the case of data-dependent communication. We will thus skip the latter versions of hs with scalar with array data type.

CASE 3: izz

```
struct VYAListType
{
  VYApixel    pixel[64];
};

void IZZ::main()
{
  struct VYAListType  Cin, Cout;

  while (true)
  {
      read(CinP, Cin);
      for (unsigned int i=0; i<64; i++)
      {
           Cout.pixel[zigzag[i]] =  Cin.pixel[i];
      }
      write(CoutP, Cout);
  }
}
```

Evaluation: From the three examples, we can see that the use of the combined scalar with array data type synchronisation and data transfer

+ needs less lines of code, less loops, which result in less code complexity;

– is very difficult to be implemented when there is data-dependent communication behaviour. A complex algorithm is needed in dealing with the indexes of the values in the communication array data type and in the computation context. So it has less applicable cases;

– needs more design effort because there are more complex data structures involved.

The evaluation table can be filled in Table 8.2:

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 11, 34; hs line 18, 19, 26, 39; izz line 10, 15: To send or receive a token, only one argument is needed.	1	3
2. Number of parameters needed in communication primitives	idct1d line 11, 34; hs line 18, 19, 26, 39; izz line 10, 15: Each of read and write needs only two parameters.	1	3
3. Lines of code	idct1d line 1-4; izz line 1-4: New structures of multiple tokens must be defined for one big token communication. hs line 1-5, 17, 21-28, 35-38, 40-43: Many lines of code have to be added for data dependent communication.	1	1
4. Number of loops	No extra loops are needed for multi-token operations.	1	3
5. Complexity of data structures	idct1d line 1-4; hs line 1-4; izz line 1-4: Complex structures have to be defined for communication.	2	1
6. Code readability	idct1d line 12-17, 30-33; izz line 13: Accessing structure members makes code less simpler than the last scenario. hs line 1-5, 17, 21-28, 35-38, 40-43: When there is data dependent communication, to keep track on the current processing point in the loaded big token, a complex algorithm is required. It is mixed with the computation algorithm, which makes code very difficult to read.	3	2
7. Number of applicable cases	It is not suitable for data dependent communication.	1	1
8. Error-proneness	izz line 13: The use of structure with array data type mixed with array operation in the computation model slightly tends to cause errors.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 11, 34; hs line 18, 19, 26, 39; izz line 10, 15: For each communication session, local memory size of multiple tokens is required.	3	2
Final Score:			40

Table 8.2: Evaluation of Combined Scalar with Array Data Type Synchronisation and Data Transfer

8.3 Combined Vector Synchronisation and Data Transfer

Description: A vector operation primitive has one more parameter than the corresponding scalar operation primitive. In the case of combined synchronisation and data transfer, the interfaces are defined as: read(c, v, n) and write(c, v, n), where c is the channel, v is a vector of values to be transferred, n is the number of tokens.

Suggested Solution: Most of the communication behaviour is described with vector operations in the original YAPI code, when multiple tokens are to be transferred through a channel. Here the main change is in hs, where a vector of pixels are inputted before the data processing and outputted to a channel afterwards.

CASE 1: idct1d

```
void idct(VYApixel *Y)
{
  ...
}

void IDCT1D::main()
{
      VYApixel Y[8];
      VYApixel z1[8], z2[8], z3[8];

      while (true)
      {
      /* The amount of 8 tokens space is needed, but all the tokens
          are not necessarily be allocated continuously in the memory. */
        read(CIn, Y, 8);
       idct(Y);
        write(COut, Y, 8);
      }
}
```

CASE 2: hs

```
void HS::main()
{
  while (true)
  {
      VYApixel        pixel;
      VYAlineLength   inLineLength;
      VYAlineLength   outLineLength;
      unsigned int        scaleFactor;

      read(inLineLengthInP, inLineLength);
       read(outLineLengthInP, outLineLength);

      scaleFactor = ceil_div(outLineLength,inLineLength);

      VYApixel* inLine = new VYApixel[inLineLength];
      VYApixel* outLine = new VYApixel[outLineLength];

      /* A vector of data is read instead of one by one in the loop. */
       read(CinP, inLine, inLineLength);
      for (unsigned int i=0; i<inLineLength; i++)
      {
          unsigned int m = i*scaleFactor;
          unsigned int n = min(m+scaleFactor,outLineLength);
          for (unsigned int j=m; j<n; j++)
```

```
            {
                outLine[j] = inLine[i];
            }
        }
        /* A vector of data is written instead of one by one in the loop. */
        write(CoutP, outLine, outLineLength);

        delete [] inLine;
        delete [] outLine;
    }
}
```

CASE 3: izz

```
void IZZ::main()
{
  VYApixel    Cin[64];
  VYApixel    Cout[64];

  while (true)
  {
      read(CinP, Cin, 64);
      for (unsigned int i=0; i<64; i++)
      {
          Cout[zigzag[i]] = Cin[i];
      }
      write(CoutP, Cout, 64);
  }
}
```

CASE 4: filter

```
void Filter::main()
{
  int j, k;
  int coeff = 1;
  bit s[2];

  int* v = new int [npixels];

  while (true)
  {
      s = select(in, cof);
      if (s[1]== 1)
      {
          read(cof, coeff);
      }

      for (j=0; j<nlines; j++)
      {
          read(in, v, npixels);
          for (k=0; k<npixels; k++)
          {
             write(out, coeff*v(k));
          }
      }
  }
  delete [] v;
}
```

Evaluation: We conclude the combined vector synchronisation and data transfer as following:

+ Compared to multiple scalar operations, it makes programming easier and code complexity less because of the reduced lines of code and number of loops when there are multiple tokens waiting to be transferred.

+ Not like the combined scalar with array data type, the combined vector synchronisation and data transfer does not require the application designer to define a new data type. So the design effort that is required is also less.

– In case it is a single token operation, to use combined scalar synchronisation and data transfer is easier.

Summary: This can be summarised in Table 8.3:

8.4 Conclusions

With comparable high ranks for scalar and vector operation on combined synchronisation and data transfer, we suggest to support combined synchronisation and data transfer and support both scalar and vector operations. It is easier to use scalar operation for single token communication and vector for multi-token communication. Unless the performance study shows much advantage from scalar with array data type operation over vector operation, vector operation is recommended for the sake of ease of programming when multiple tokens are to be transferred. Having discussed about combined synchronisation and data transfer in this chapter, we are going to look into separate synchronisation and data transfer in the next chapters. According to the way that data transfer is performed, separate synchronisation can be performed directly or indirectly. The coming chapter is about direct access.

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 10, 12; hs line 14, 24; izz line 7, 12: filter line 12, 19: To send or receive multiple tokens, only one argument is needed.	1	3
2. Number of parameters needed in communication primitives	idct1d line 10, 12; hs line 14, 24; izz line 7, 12: filter line 12, 19: Each of read and write needs three parameters.	1	2
3. Lines of code	Multiple tokens can be sent or received with one line of code.	1	3
4. Number of loops	No extra loop is needed for communication.	1	3
5. Complexity of data structures	idct1d line 6; hs line 12, 13; izz line 3, 4; filter 6: Array data types are used for multi-token communication.	2	2
6. Code readability	idct1d line 10, 12; hs line 14, 24; izz line 7, 12: filter line 12, 19: Communication part is very compact and clear.	3	3
7. Number of applicable cases	It is suitable for any communication behaviour type: data dependent or independent.	1	3
8. Error-proneness	No pointers and difficult data structures are used. Change of one part of code does not affect the other part.	3	3
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 10, 12; hs line 14, 24; izz line 7, 12: filter line 12, 19: For each communication session, local memory size of multiple tokens is required.	3	2
Final Score:			51

Table 8.3: Evaluation of Combined Vector Synchronisation and Data

9 Direct Access

In this chapter, ease of programming on direct access will be studied. Direct access means accessing a token with a memory location address which is obtained from an acquire function. Direct Access makes data transfers more efficient and makes it possible to only transfer parts of a token.

The main issue here is the design choice between in-order intra-task synchronisation and data transfer and out-of-order in-order intra-task synchronisation and data transfer. Besides that the comparison between vector operation and scalar with array data type is further studied. In order to do that, three scenarios are created in three sections. They are separate, vector, in-order intra-task synchronisation and direct access in Section 9.1, separate, scalar with array data type, in-order intra-task synchronisation and direct access in Section 9.2 and separate, out-of-order intra-task synchronisation and direct access in Section 9.3.

9.1 Separate, Vector, In-Order Intra-task Synchronisation and Direct Access

Description: With separate synchronisation and data transfer, a communication act is realised with three primitives, namely acquireData/Room, load/store and release-Room/Data. In-order intra-task means that within a task, a token is released in the same order as the token is acquired before.

Suggested Solution: Define an array of pointers of the token type. Each element of the array points to a token. The contents of the array are filled in by an acquire primitive.

CASE 1: idct1d

```
void IDCT1D::main()
{
     VYApixel z1[8], z2[8], z3[8];
    /* Define an array of pointers of the token type: VYApixel. */
    VYApixel* r[8];

     while (true)
     {
            /* Acquire 8 full tokens from channel CIn, and store the identifiers
          in the vector r of length 8. */
          /* When token size is big, it is more memory efficient to use
          identifiers. If the token size is small, it is more memory efficient
          to get  tokens themselves, as in combined synchronisation and
          data transfer. */
        acquireData(CIn, r, 8);
```

```
            // Stage 1:
              /* Access the tokens by accessing the memory locations where
            the elements of r point to. */
           /* Pointer operations make data structure a bit more complex,
            code less readable and more error-prone. */
            but( *(r[0]), *(r[4]), z1[1], z1[0]);
            z1[2] = SUB(CMUL( 8867,  *(r[2])), CMUL(21407,  *(r[6])));
            z1[3] = ADD(CMUL(21407,  *(r[2])), CMUL( 8867,  *(r[6])));
            but(*(r[1]), *(r[7]), z1[4], z1[7]);
            z1[5] = CMUL(23170, *(r[3]));
            z1[6] = CMUL(23170, *(r[5]));
            /* Release the 8 old acquired tokens. */
            releaseRoom(CIn, 8);

            // Stage 2:
            but(z1[0], z1[3], z2[3], z2[0]);
            but(z1[1], z1[2], z2[2], z2[1]);
            but(z1[4], z1[6], z2[6], z2[4]);
            but(z1[7], z1[5], z2[5], z2[7]);

            // Stage 3:
            z3[0] = z2[0];
            z3[1] = z2[1];
            z3[2] = z2[2];
            z3[3] = z2[3];
            z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
            z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
            z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
            z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

            // Stage 4:
              /* Acquire 8 empty tokens from channel COut, and store the
            identifiers in the vector r of length 8. */
            acquireRoom(COut, r, 8);
            but(z3[0], z3[7], *(r[7]), *(r[0]));
            but(z3[1], z3[6], *(r[6]), *(r[1]));
            but(z3[2], z3[5], *(r[5]), *(r[2]));
            but(z3[3], z3[4], *(r[4]), *(r[3]));

          /* Release the old acquired 8 tokens. */
            releaseData(COut, 8);
    }
}
```

CASE 2: hs

```
void HS::main()
{
  while (true)
  {
      VYAlineLength   inLineLength;
      VYAlineLength   outLineLength;
      unsigned int        scaleFactor;
      VYALineLength*  r;
      acquireData(inLineLengthInP, r);
      inLineLength = *r;
      releaseRoom(inLineLengthInP);
      acquireData(outLineLengthInP, r);
      outLineLength = *r;
      releaseRoom(outLineLengthInP);

      scaleFactor = ceil_div(outLineLength,inLineLength);

      VYApixel *rin = new VYApixel[inLineLength];
      VYApixel *rout = new VYApixel[outLineLength];

      acquireData(CinP, rin, inLineLength);
      acquireRoom(CoutP, rout, outLineLength);
       for (unsigned int i=0; i<inLineLength; i++)
       {
           unsigned int m = i*scaleFactor;
           unsigned int n = std::min(m+scaleFactor,outLineLength);
           for (unsigned int j=m; j<n; j++)
           {
              *(rout[j]) = *(rin[i]);
           }
       }
       releaseRoom(CinP, inLineLength);
      releaseData(CoutP, outLineLength);

      delete [ ] rin;
      delete [ ] rout;
  }
}
```

CASE 3: izz

```
void IZZ::main()
{
 VYApixel    * rin[64];
 VYApixel    * rout[64];

  while (true)
  {
     acquireData(CinP, rin, 64);
     acquireRoom(CoutP, rout,  64);

       for (unsigned int i=0; i<64; i++)
       {
          *(rout[zigzag[i]]) =*(rin[i]);
       }
     releaseRoom(CinP, 64);
     releaseData(CoutP, 64);
  }
}
```

CASE 4: filter

```
void Filter::main()
{
  int j, k;
  int coeff = 1;
  bit s[2];
  int *r;
  int *rin[npixels];
  int *rout[npixels];

  int* v = new int [npixels];

  while (true)
  {
     s = select(in, cof);
      if ( s[1]== 1)
      {
        acquireData(cof, r);
        coeff = *r;
        releaseRoom(cof);
      }

      for (j=0; j<nlines; j++)
      {
        acquireData(in, rin, npixels);
        acquireRoom(out, rout, npixels);

          for (k=0; k<npixels; k++)
          {
           rout[k] =  coeff*rin[k];
          }
        releaseRoom(in, npixels);
        releaseData(out, npixels);
      }
      delete [] v;
  }
}
```

Evaluation: To finish a send or receive data operation, the separate synchronisation

– requires at least two times as many primitives as the combined synchronisation does;

Note that for code readability, it is better to acquire all the data and room needed in the beginning and release them at the end. However, for the sake of memory usage, we suggest to acquire data or room as late as possible and release them as early as possible.

Summary: With the criteria we defined in an early chapter, we can rank separate vector in-order intra-task synchronisation and data transfer as in Table 9.1:

Criteria	Observation	Weight Factors	Scores
1. Number of communication primitives needed	idct1d line 7, 14, 27, 32; hs line 9, 14, 18, 19, 29, 30; izz line 7, 8, 13, 14; filter line 15, 17, 21, 22, 27, 28: To send or receive a token, two arguments are needed.	1	2
2. Number of parameters needed in communication primitives	idct1d line 7, 14, 27, 32; hs line 18, 19, 29, 30; izz line 7, 8, 13, 14; filter line 15, 17, 21, 22, 27, 28: For multiple token operations, each acquire needs only three parameters and each release needs one parameter.	1	2
3. Lines of code	idct1d line 7-14, 27-32; hs line 9-14, 18-30; izz line 7-14; filter line: 15-17, 21-28: Besides the acquire and release, data transfers can be combined with the computation part.	1	2
4. Number of loops	No extra loops are needed.	1	3
5. Complexity of data structures	idct1d line 4; hs line 8, 16, 17; izz line 3, 4; filter line 6, 7, 8: Array of pointer types have to be used for communication.	2	1
6. Code readability	idct1d line 8-13, 28-31; hs line 10, 13, 26; izz line 11; filter line 25: To dereference tokens via pointers in arrays makes code less readable than to use vector operation.	3	2
7. Number of applicable cases	It is suitable for any communication behaviour type: data dependent or independent.	1	3
8. Error-proneness	The use of pointers in arrays tends to cause errors. But the change of one part of code does not affect the other part.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 4; hs line 8, 16, 17; izz line 3, 4; filter line 6, 7, 8: For each communication session, only local memory size of multiple pointers is required. This is efficient when the tokens are big. But for small tokens it is costly.	3	2
Final Score:			41

Table 9.1: Evaluation of Separate, Vector, In-Order Intra-task Synchronisation and Direct Access

9.2 Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Direct Access

Description: The difference between this scenario and the previous one is the data structure. Here we use array data type.

Suggested Solution: Define a structure that contains an array of the original token type. As in the previous section, the addresses of the tokens are filled into the array by the acquire primitives.

CASE 1: idct1d

```
/* Define a structure containing an array of the original  token type.
   The structure is now the new token type. */
struct VYABlock˙type

  VYApixel Y[8];
;

void IDCT1D::main()
{
      VYApixel        z1[8], z2[8], z3[8];

      while (true)
      {
        const VYABlock_type*  VYAPointer1;

         /* Acquire a big full token from channel CIn and store
          the address of the big token in VYAPointer1. */
         /* Only one pointer is required from the local memory. */
         acquireData(CIn, VYAPointer1);

       /* Access the original small tokens by accessing the elements
          of the big token. */
       /* Pointer operations make data structure a bit more complex,
          code less readable and more error-prone. */
        // Stage 1:
        but(VYAPointer1->Y[0], VYAPointer1->Y[4]),
              z1[1], z1[0]);
        z1[2] = SUB(CMUL( 8867, VYAPointer1->[2]),
              CMUL(21407, VYAPointer1->Y[6]));
        z1[3] = ADD(CMUL(21407, VYAPointer1->Y[2]),
              CMUL( 8867, VYAPointer1->Y[6]));
        but(VYAPointer1->Y[1], VYAPointer1->Y[7], z1[4], z1[7]);
        z1[5] = CMUL(23170, VYAPointer1->Y[3]);
        z1[6] = CMUL(23170, VYAPointer1->Y[5]);
        /* Release the old acquired token. */
         releaseRoom(CIn);

        // Stage 2:
        but(z1[0], z1[3], z2[3], z2[0]);
        but(z1[1], z1[2], z2[2], z2[1]);
        but(z1[4], z1[6], z2[6], z2[4]);
        but(z1[7], z1[5], z2[5], z2[7]);

        // Stage 3:
        z3[0] = z2[0];
        z3[1] = z2[1];
        z3[2] = z2[2];
        z3[3] = z2[3];
        z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
        z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
        z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
        z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

        // Stage 4:
         VYABlock_type* VYAPointer2;

        /* Acquire a big empty  token from channel COut and store
         the address of the big token in VYAPointer1. */
        acquireRoom(Cout, VYAPointer2);
        but(z3[0], z3[7], VYAPointer2->Y[7], VYAPointer2->Y[0]);
        but(z3[1], z3[6], VYAPointer2->Y[6], VYAPointer2->Y[1]);
        but(z3[2], z3[5], VYAPointer2->Y[5], VYAPointer2->Y[2]);
        but(z3[3], z3[4], VYAPointer2->Y[4], VYAPointer2->Y[3]);

       /* Release the old acquired token. */
        releaseData(COut);
       }
}
```

CASE 2: hs

The example for Combined Synchronisation and Data Transfer with Array Data Type has shown us the difficulties in using array data type in hs, which is the case of data-dependent communication. So here we skip this version.

CASE 3: izz

```
struct VYAListType
{
  VYApixel    pixel[64];
};

void IZZ::main()
{
  while (true)
  {
      const struct VYAListType* Cin;
       struct VYAListType* Cout;

       acquireData(CinP, Cin);
       acquireRoom(CoutP, Cout);
      for (unsigned int i=0; i<64; i++)
      {
          Cout->pixel[zigzag[i]] = Cin->pixel[i];
      }
       releaseRoom(CinP);
       releaseData(CoutP);
  }
}
```

Evaluation: Compared to Separate, Vector, In-Order Intra-task Synchronisation and Direct Access, Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Direct Access

– is less easy to program because the data structure is more complicated;

+ only needs to acquire and release one token instead of a vector of tokens.

Criteria	Observation	Weight Factors	Scores
1. Number of communication primitives needed	idct1d line 11, 18, 32, 37; izz line 11, 12, 17, 18: To send or receive a token, two parameters are needed.	1	2
2. Number of parameters needed in communication primitives	idct1d line 11, 18, 32, 37; izz line 11, 12, 17, 18: Each acquire needs only two arguments and each release needs one argument.	1	3
3. Lines of code	idct1d line 11-18; izz line 11-18: Besides the acquire and release, data transfer can be combined into computation.	1	2
4. Number of loops	No extra loops are needed.	1	3
5. Complexity of data structures	idct1d line 10; izz line 10: Pointer of structure types have to be used for communication.	2	1
6. Code readability	idct1d line 12-17; izz line 15: To dereference tokens via pointers to structures makes code less readable than to use vector operation.	3	2
7. Number of applicable cases	It is not suitable for data dependent communication.	1	1
8. Error-proneness	The use of pointers to structures tends to cause errors. But the change of one part of code does not affect the other part.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 10; izz line 10: For each communication session, only local memory size of one pointers is required.	3	3
Final Score:			43

Table 9.2: Evaluation of Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Direct Access

Summary: We rank separate, scalar with array data type, in-order intra-task synchronisation and data transfer as in Table 9.2:

9.3 Separate, Out-of-Order Intra-task Synchronisation and Direct Access

Description: Out-of-order intra-task synchronisation means to acquire and release a token (full or empty) in a different order within a task. Not like in-order intra-task synchronisation, the order of release primitives do not have to stick to the order of the corresponding acquire primitives.

Suggested Solution: Release a token once it is not needed any more.

CASE 1: idct1d

```
1 void IDCT1D::main()
```

```
{
        VYApixel z1[8], z2[8], z3[8];
        VYApixel* r[8];

        while (true)
        {
            acquireData(CIn, r, 8);

            // Stage 1:
              but(*(r[0]), *(r[4]), z1[1], z1[0]);
           /* Release the  tokens that are no longer needed. */
           /* Releasing the tokens earlier makes the memory in the channel
            available earlier to later communication.
             releaseRoom(Cin, r[0]);
              releaseRoom(Cin, r[4]);
            z1[2] = SUB(CMUL( 8867, *(r[2])), CMUL(21407, *(r[6])));
            z1[3] = ADD(CMUL(21407, *(r[2])), CMUL( 8867, *(r[6])));
            releaseRoom(Cin, r[2]);
             releaseRoom(Cin, r[6]);
            but(*(r[1]), *(r[7]), z1[4], z1[7]);
            releaseRoom(Cin, r[1]);
             releaseRoom(Cin, r[7]);
            z1[5] = CMUL(23170, *(r[3]));
            z1[6] = CMUL(23170, *(r[5]));
            releaseRoom(Cin, r[3]);
            releaseRoom(Cin, r[5]);

            // Stage 2:
            but(z1[0], z1[3], z2[3], z2[0]);
            but(z1[1], z1[2], z2[2], z2[1]);
            but(z1[4], z1[6], z2[6], z2[4]);
            but(z1[7], z1[5], z2[5], z2[7]);

            // Stage 3:
            z3[0] = z2[0];
            z3[1] = z2[1];
            z3[2] = z2[2];
            z3[3] = z2[3];
            z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
            z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
            z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
            z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

            // Stage 4:
            acquireRoom(Cout, r, 8);
            but(z3[0], z3[7], *(r[7]), *(r[0]));
            releaseData(Cout, r[7]);
             releaseData(Cout, r[0]);
            but(z3[1], z3[6], *(r[6]), *(r[1]));
            releaseData(Cout, r[6]);
            releaseData(Cout, r[1]);
            but(z3[2], z3[5], *(r[5]), *(r[2]));
            releaseData(Cout, r[5]);
            releaseData(Cout, r[2]);
            but(z3[3], z3[4], *(r[4]), *(r[3]));
            releaseData(Cout, r[4]);
            releaseData(Cout, r[3]);
  }
}
```

CASE 2: hs

```
void HS::main()
{
  while (true)
  {
      VYApixel        pixel;
      VYAlineLength   inLineLength;
      VYAlineLength   outLineLength;
      unsigned int        scaleFactor;
      VYALineLength*  r;

     acquireData(inLineLengthInP, r);
     inLineLength = *r;
     releaseRoom(inLineLengthInP, r);
     acquireData(outLineLengthInP, r);
     outLineLength = *r;
     releaseRoom(outLineLengthInP, r);

      scaleFactor = ceil_div(outLineLength,inLineLength);

     VYApixel *rin = new VYApixel[inLineLength];
     VYApixel *rout = new VYApixel[outLineLength];

     acquireData(CinP, rin, inLineLength);
     acquireRoom(CoutP, rout, outLineLength);
      for (unsigned int i=0; i<inLineLength; i++)
      {
          unsigned int m = i*scaleFactor;
          unsigned int n = std::min(m+scaleFactor,outLineLength);
          for (unsigned int j=m; j<n; j++)
          {
           *(rout[j]) = *(rin[i]);
           releaseRoom(CinP, rin[i]);
           releaseData(CoutP, rout[j]);
          }
      }

      delete [ ] rin;
      delete [ ] rout;
  }
}
```

CASE 3: izz

```
void IZZ::main()
{
  VYApixel    * rin[64];
  VYApixel    * rout[64];
while (true)
{
   acquireData(CinP, rin, 64);
   acquireRoom(CoutP, rout, 64);

     for (unsigned int i=0; i<64; i++)
     {
       *(rout[zigzag[i]]) =*(rin[i]);
       releaseRoom(CinP, rin[i]);
       releaseData(CoutP, rout[i]);
     }
 }
}
```

Evaluation: Out-of-Order intra-task synchronisation

+ can relieve the memory usage problem especially when large blocks of data is transferred.

– needs more parameters to be defined compared to the same operation (scalar or vector) in in-order data synchronisation.

Summary: The ranking is as in Table 9.3:

9.4 Conclusions

The comparison suggests that in-order vector, in-order scalar with array data type and out-of order synchronisation and direct access are generally equal in the sense of ease of programming. They all have the strength of no extra loops and high code reusability and the weakness of requiring complex data structures. One remarkable advantage of in-order scalar with array data type synchronisation and direct access is that it has very high local memory space efficiency. Even for each multiple token communication session, only local memory size of one pointers is needed. But the drawback is that it is not suitable for data dependent communication behaviours. From the memory usage perspective, out-of order synchronisation and data transfer is better than in-order synchronisation and data transfer because it allows the tasks to release the unneeded channel memory as soon as possible.

As we shown in this chapter, with direct access, data transfer is performed by accessing a token with a memory location address directly. In the next chapter, we will present another way of data access: indirect access, for which explicit system calls are introduced to accomplish data access.

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 7, 9, 10, 13, 14, 16, 17, 20, 21, 34, 36, 37, 39, 40, 42, 43; hs line 10, 12, 13, 15, 19, 20, 28, 29; izz line 7, 8, 12, 13; To send or receive a token, two primitives are needed.	1	2
2. Number of parameters needed in communication primitives	idct1d line 7, 9, 10, 13, 14, 16, 17, 20, 21, 34, 36, 37, 39, 40, 42, 43; hs line 10, 12, 13, 15, 19, 20, 28, 29; izz line 7, 8, 12, 13: For multiple token operations, each acquire needs only three parameters. For each single token operation, each release needs two parameters (three for multiple tokens).	1	1
3. Lines of code	idct1d line 7, 9, 10, 13, 14, 16, 17, 20, 21, 34, 36, 37, 39, 40, 42, 43; hs line 10, 12, 13, 15, 19, 20, 28, 29; izz line 7, 8, 12, 13; If the release is done in a fine grain as in the examples, there will be many lines extra.	1	2
4. Number of loops	No extra loops are needed.	1	3
5. Complexity of data structures	idct1d line 4; hs line 9, 17, 18; izz line 3, 4: Array of pointer types have to be used for communication.	2	1
6. Code readability	idct1d line 8, 11, 12, 15, 18, 19, 35, 38, 41, 44; hs line 11, 14, 27; izz line 11: To dereference tokens via pointers in arrays makes code less readable than to use vector operation.	3	2
7. Number of applicable cases	It is suitable for any communication behaviour type: data dependent or independent.	1	3
8. Error-proneness	The use of pointers in arrays tends to cause errors. But the change of one part of code does not affect the other part.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 4, 7, 9, 10, 13, 14, 16, 17, 20, 21, 34, 36, 37, 39, 40, 42, 43; hs line 9, 10, 12, 13, 15, 17, 18, 19, 20, 28, 29; izz line 7, 8, 11, 12, 13: For each communication session, only local memory size of multiple pointers is required. This is efficient when the tokens are big, for small tokens it is still costly. But the demands for global memory can be alleviated by releasing the unneeded FIFO memory earlier.	3	3
Final Score:			43

Table 9.3: Evaluation of Separate, Out-of-Order Intra-task Synchronisation and Direct Access

10 Indirect Access

With indirect access, data transfer is explicitly done via functions, namely load and store. The reason to consider indirect access is because it can more clearly separate computation and communication and provides an efficient solution for small tokens because address generation can be hidden in the infrastructure thus consecutive tokens in a stream might not be stored in consecutive memory locations. Indirect access can take two forms: absolute and relative token references. Brief descriptions of them are addressed in the coming sections.

The task of this chapter is to assess indirect access with both absolute and relative token references, compare them against each other and at the same time also compare indirect access with direct access. The chapter is organised as follows: We discuss separate, vector, in-order intra-task synchronisation and indirect access with absolute references in Section 10.1; separate, vector, in-order intra-task synchronisation and indirect access with relative references in Section 10.2 and separate, scalar with array data type, in-order intra-task synchronisation and indirect Access with relative references in Section 10.3 and summarise them in Section 10.4.

10.1 Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Absolute References

Description: An absolute token reference refers to the same token until the token is released.

Suggested Solution: Define an array (for multiple token cases) or a variable (for single token cases) of the reference type, obtain the value(s) of the reference(s) from the acquireData/Room primitives, transfer the tokens from the channels to the local variables with load/store and release them at the end.

CASE 1: idct1d

```
void IDCT1D::main()
{
      VYApixel z1[8], z2[8], z3[8];
      VYApixel y1, y2;
      /* Define an array with the reference type. */
     Reference r[8];

      while (true)
      {
              acquireData(CIn, r, 8);
```

```
    // Stage 1:
    /* Load the first token just acquired to the local variable y1. */
     load(Cin, r[0], y1);
 /* Load the 5th token just acquired to the local variable y1. */
     load(Cin, r[4], y2);
    but(y1, y2, z1[1], z1[0]);

     load(Cin, r[2], y1);
    load(Cin, r[6], y2);
    z1[2] = SUB(CMUL( 8867, y1), CMUL(21407, y2));
    z1[3] = ADD(CMUL(21407, y1)), CMUL( 8867, y2));

     load(Cin, r[1], y1);
     load(Cin, r[7], y2);
    but(y1, y2, z1[4], z1[7]);

     load(Cin, r[3], y1);
     load(Cin, r[5], y2);
    z1[5] = CMUL(23170, y1);
    z1[6] = CMUL(23170, y2);
     releaseRoom(CIn, 8);

    // Stage 2:
    but(z1[0], z1[3], z2[3], z2[0]);
    but(z1[1], z1[2], z2[2], z2[1]);
    but(z1[4], z1[6], z2[6], z2[4]);
    but(z1[7], z1[5], z2[5], z2[7]);

    // Stage 3:
    z3[0] = z2[0];
    z3[1] = z2[1];
    z3[2] = z2[2];
    z3[3] = z2[3];
    z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
    z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
    z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
    z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

    // Stage 4:
    acquireRoom(Cout, r, 8);
    but(z3[0], z3[7], y1, y2);
    store(Cin, r[7], y1);
    store(Cin, r[0], y2);

    but(z3[1], z3[6], y1, y2);
    store(Cin, r[6], y1);
    store(Cin, r[1], y2);

    but(z3[2], z3[5], y1, y2);
    store(Cin, r[5], y1);
    store(Cin, r[2], y2);

    but(z3[3], z3[4], y1, y2);
    store(Cin, r[4], y1);
    store(Cin, r[3], y2);
```

```
            releaseData(COut, 8);
        }
}
```

CASE 2: hs

```
void HS::main()
{
  while (true)
  {
      VYApixel        pixel;
      VYAlineLength   inLineLength;
      VYAlineLength   outLineLength;
      unsigned int        scaleFactor;
     Reference      r;
     Reference      r0;

     acquireData(inLineLengthInP, r);
     load(inLineLengthInP, r, inLineLength);
     releaseRoom(inLineLengthInP);

     acquireData(outLineLengthInP);
     load(outLineLengthInP, r, outLineLength);
     releaseRoom(outLineLengthInP);

      scaleFactor = ceil_div(outLineLength,inLineLength);

      for (unsigned int i=0; i<inLineLength; i++)
      {
        acquireData(CinP, r);
        load(CinP, r, pixel);
          unsigned int m = i*scaleFactor;
          unsigned int n = std::min(m+scaleFactor,outLineLength);
          for (unsigned int j=m; j<n; j++)
          {
           acquireRoom(CoutP, r0);
           store(CoutP, r0, pixel);
           releaseData(CoutP);
          }
        releaseRoom(CinP);
      }
  }
}
```

CASE 3: izz

```
void IZZ::main()
{
  VYApixel    Cin;
 Reference   rin[64];
 Reference    rout[64];

  while (true)
  {
    acquireData(CinP, rin, 64);
    acquireRoom(CoutP, rout, 64);

     for (unsigned int i=0; i<64; i++)
     {
       load(CinP, rin[i], Cin);
       store(CoutP, rout[zigzag[i]], Cin);
     }
    releaseRoom(CinP, 64);
    releaseData(CoutP, 64);
  }
}
```

Evaluation: Compared to direct references, absolute access with indirect references:

– needs more operations to send or receive data from the channel;

+ does not allow the designer to access the data directly via pointer operation, which can reduce the chances of memory access error.

Summary: We rank the scenario as in Table 10.1:

10.2 Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Relative References

Description: A relative token reference is an integer value that addresses the token with the given integer distance relative to the oldest token that has been acquired but not yet been released.

Suggested Solution: Keep track on the number of tokens that have been acquired but not yet been released.

CASE 1: idct1d

In embedded system design, software developers often must take the amount of local memory available in the system into account. To show the impact on the application design, here we will give two code examples, each of which is suitable for the case of sufficient or insufficient local memory. The two examples at the same time also show the effects of that on using relative references.

idct1d(1):

If sufficient local memory is available for the processor, it is possible to load a vector of data at one time.

```
void IDCT1D::main()
{
     VYApixel Y[8];
     VYApixel z1[8], z2[8], z3[8];
```

Criteria	Observation	Weight Factors	Scores
1. Number of communication primitives needed	idct1d line 8-13, 16-17, 19-20, 23, 36, 38-42, 44-45, 47-49; hs line 11-16, 20-21, 26-28, 30; izz line 8-9, 12-13, 15-16: To send or receive a token, three primitives are needed.	1	1
2. Number of parameters needed in communication primitives	idct1d line 8-13, 16-17, 19-20, 23, 36, 38-42, 44-45, 47-49; hs line 11-16, 20-21, 26-28, 30; izz line 8-9, 12-13, 15-16: For multiple token operations, each acquire, data transfer or release needs three arguments.	1	2
3. Lines of code	idct1d line 8-13, 16-17, 19-20, 23, 36, 38-42, 44-45, 47-49; hs line 11-16, 20-21, 26-28, 30; izz line 8-9, 12-13, 15-16: A clear separation of the computation part and the communication part leads to more lines of code.	1	1
4. Number of loops	No extra loops are needed.	1	3
5. Complexity of data structures	idct1d line 5; hs line 9, 10; izz line 4, 5: An array data type is used for multiple token communication.	2	2
6. Code readability	idct1d line 9-23, 37-48; hs line 12, 15, 21, 27; izz line 12, 13: A clear separation of computation and communication parts brings more code readability.	3	3
7. Number of applicable cases	It is suitable for any communication behaviour type: data dependent or independent.	1	3
8. Error-proneness	The use of pointers can be avoided if a reference type is defined. Change of one part of code does not affect the other part.	3	3
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 5; hs line 9, 10; izz line 4, 5: For each communication session, only local memory size of multiple references is required. This is efficient when the tokens are big. But for small tokens it is costly.	3	2
Final Score:			47

Table 10.1: Evaluation of Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Absolute References

```
5       while (true)
6       {
7           acquireData(CIn, 8);
            /* The distance relative to the oldest token
            that has been acquired but
            not yet been released is 0. */
8           load(CIn, 0, Y, 8);
9           releaseRoom(CIn, 8);
```

```
        // Stage 1:
        but(Y[0], Y[4], z1[1], z1[0]);
        z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
        z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
        but(Y[1], Y[7], z1[4], z1[7]);
        z1[5] = CMUL(23170, Y[3]);
        z1[6] = CMUL(23170, Y[5]);

        // Stage 2:
        but(z1[0], z1[3], z2[3], z2[0]);
        but(z1[1], z1[2], z2[2], z2[1]);
        but(z1[4], z1[6], z2[6], z2[4]);
        but(z1[7], z1[5], z2[5], z2[7]);

        // Stage 3:
        z3[0] = z2[0];
        z3[1] = z2[1];
        z3[2] = z2[2];
        z3[3] = z2[3];
        z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
        z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
        z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
        z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

        // Stage 4:
        but(z3[0], z3[7], Y[7], Y[0]);
        but(z3[1], z3[6], Y[6], Y[1]);
        but(z3[2], z3[5], Y[5], Y[2]);
        but(z3[3], z3[4], Y[4], Y[3]);

         acquireRoom(COut, 8);
        /* The distance relative to the oldest token
           that has been acquired but
           not yet been released is 0. */
         store(COut, 0, Y, 8);
         releaseData(COut, 8);
    }
}
```

idct1d(2):
If there is insufficient local memory, one must load a small amount of data every time.

```
void IDCT1D::main()
{
    VYApixel y1, y2;
    VYApixel z1[8], z2[8], z3[8];

    while (true)
    {
     acquireData(CIn, 8);

        // Stage 1:
      /* Load the token with the distance relative to
           the oldest token that has been acquired but
           not yet been released of 0. */
        load(CIn, 0, y1);
    /* Load the token with the distance relative
       to the oldest token that has been acquired
       but not yet been released of 4. */
       load(CIn, 4, y2);
       but(y1, y2, z1[1], z1[0]);

       load(CIn, 2, y1);
       load(CIn, 6, y2);
       z1[2] = SUB(CMUL( 8867, y1), CMUL(21407, y2));
       z1[3] = ADD(CMUL(21407, y1), CMUL( 8867, y2));

       load(CIn, 1, y1);
       load(CIn, 7, y2);
       but(y1, y2, z1[4], z1[7]);

       load(CIn, 3, y1);
       load(CIn, 5, y2);
       z1[5] = CMUL(23170, y1);
       z1[6] = CMUL(23170, y2);

       releaseRoom(CIn, 8);

       // Stage 2:
       but(z1[0], z1[3], z2[3], z2[0]);
       but(z1[1], z1[2], z2[2], z2[1]);
       but(z1[4], z1[6], z2[6], z2[4]);
       but(z1[7], z1[5], z2[5], z2[7]);

       // Stage 3:
       z3[0] = z2[0];
       z3[1] = z2[1];
       z3[2] = z2[2];
       z3[3] = z2[3];
       z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
       z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
       z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
       z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

       // Stage 4:
       acquireRoom(COut, 8);

       but(z3[0], z3[7], y1, y2);
       store(COut, 7, y1);
        store(COut, 0, y2);
       but(z3[1], z3[6], y1, y2);
       store(COut, 6, y1);
       store(COut, 1, y2);
       but(z3[2], z3[5], y1, y2);
       store(COut, 5, y1);
        store(COut, 2, y2);
       but(z3[3], z3[4], y1, y2);
       store(COut, 4, y1);
        store(COut, 3, y2);
```

```
          releaseData(COut, 8);
      }
}
```

CASE 2: hs

```
void HS::main()
{
  while (true)
  {
      VYApixel        pixel;
      VYAlineLength   inLineLength;
      VYAlineLength   outLineLength;
      unsigned int        scaleFactor;

     acquireData(inLineLengthInP);
     load(inLineLengthInP, 0, inLineLength);
     releaseRoom(inLineLengthInP);

     acquireData(outLineLengthInP);
     load(outLineLengthInP, 0, outLineLength);
     releaseRoom(outLineLengthInP);

      scaleFactor = ceil_div(outLineLength,inLineLength);

     acquireData(CinP, inLineLength);
     acquireRoom(CoutP, outLineLength);
      for (unsigned int i=0; i<inLineLength; i++)
      {
        load(CinP, i, pixel);
          unsigned int m = i*scaleFactor;
          unsigned int n = std::min(m+scaleFactor,outLineLength);
          for (unsigned int j=m; j<n; j++)
          {
           store(CoutP, j, pixel);
          }
      }
     releaseRoom(CinP, inLineLength);
     releaseData(CoutP, outLineLength);
  }
}
```

CASE 3: izz

```
void IZZ::main()
{
  VYApixel    Cin;

  while (true)
  {
    acquireData(CinP, 64);
    acquireRoom(CoutP, 64);

      for (unsigned int i=0; i<64; i++)
      {
```

```
      load(CinP, i, Cin);
      store(CoutP, zigzag[i], Cin);
    }
    releaseRoom(CinP, 64);
    releaseData(CoutP, 64);
  }
}
```

Evaluation: Compared to absolute reference, relative reference:

– sometimes is difficult to use because of the needs to keep track on the relative distance to the oldest token that is acquired but not yet been released.

Summary: The scenario is ranked as in Table 10.2:

10.3 Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Indirect Access with Relative References

Description: The difference of this scenario with the last one is that here array data type is used instead of vector for multiple token communication.

Suggested Solution: A structure with an array member is defined and multiple token communication is done via one big token communication.

CASE 1: idct1d

```
struct VYABlock˙type

 VYApixel Y[8];
;

/* Inverse 1-D Discrete Cosine Transform.
   Result Y is scaled up by factor sqrt(8).
   Original Loeffler algorithm.
*/
void IDCT1D::main()
{
    VYApixel z1[8], z2[8], z3[8];
    struct VYABlock˙type VYABlock;

    while (true)
    {
      acquireData(CIn);
      load(CIn, 0, VYABlock);
      releaseRoom(CIn);

        // Stage 1:
        but(VYABlock.Y[0], VYABlock.Y[4], z1[1], z1[0]);
        z1[2] = SUB(CMUL( 8867, VYABlock.Y[2]), CMUL(21407, VYABlock.Y[6]));
        z1[3] = ADD(CMUL(21407, VYABlock.Y[2]), CMUL( 8867, VYABlock.Y[6]));
```

Criteria	Observation	Weight Factors	Scores
1. Number of communication primitives needed	idct1d(1) line 7-9, 32-34; idct1d(2): 7-9, 11-12, 15-16, 18-19, 22, 35, 37-38, 40-41, 43-44, 46-48; hs line 9-14, 16-17, 20, 25, 28, 29; izz line 6-7, 10-11, 13-14: To send or receive a token, three arguments are needed.	1	1
2. Number of parameters needed in communication primitives	idct1d(1) line 7-9, 32-34; idct1d(2): 7-9, 11-12, 15-16, 18-19, 22, 35, 37-38, 40-41, 43-44, 46-48; hs line 9-14, 16-17, 20, 25, 28, 29; izz line 6-7, 10-11, 13-14: For multiple token operations, each acquire needs three primitives, each data transfer needs four primitives and each release needs two primitive.	1	1
3. Lines of code	idct1d(1) line 7-9, 32-34; idct1d(2): 7-9, 11-12, 15-16, 18-19, 22, 35, 37-38, 40-41, 43-44, 46-48; hs line 9-14, 16-17, 20, 25, 28, 29; izz line 6-7, 10-11, 13-14: A clear separation of computation part and communication part leads to more lines of code.	1	1
1. Number of loops	No extra loops are needed.	1	3
2. Complexity of data structures	idct1d(1) line 3: An array data type is used for multiple token communication.	2	2
3. Code readability	A clear separation of computation and communication parts brings more code readability.	3	3
4. Number of applicable cases	It is suitable for any communication behaviour type: data dependent or independent.	1	3
5. Error-proneness	Change of one part of code can affect the other part due to the change of the relative distance.	3	2
6. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
7. Memory usage	idct1d(1) line 3: For each communication session, local memory size of multiple tokens is required.	3	2
Final Score:			43

Table 10.2: Evaluation of Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Relative References

```
13          but(VYABlock.Y[1], VYABlock.Y[7], z1[4], z1[7]);
14          z1[5] = CMUL(23170, VYABlock.Y[3]);
15          z1[6] = CMUL(23170, VYABlock.Y[5]);

            // Stage 2:
16          but(z1[0], z1[3], z2[3], z2[0]);
17          but(z1[1], z1[2], z2[2], z2[1]);
18          but(z1[4], z1[6], z2[6], z2[4]);
19          but(z1[7], z1[5], z2[5], z2[7]);
```

```
        // Stage 3:
        z3[0] = z2[0];
        z3[1] = z2[1];
        z3[2] = z2[2];
        z3[3] = z2[3];
        z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
        z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
        z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
        z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

        // Stage 4:
        but(z3[0], z3[7], VYABlock.Y[7], VYABlock.Y[0]);
        but(z3[1], z3[6], VYABlock.Y[6], VYABlock.Y[1]);
            but(z3[2], z3[5], VYABlock.Y[5], VYABlock.Y[2]);
        but(z3[3], z3[4], VYABlock.Y[4], VYABlock.Y[3]);

      acquireRoom(COut);
      store(COut, 0, VYABlock);
      releaseData(COut);

    }
}
```

CASE 2: hs

In this program, the number of the tokens needed to be load and store is unknown at compile time. So it is not suitable for scalar with array data type.

CASE 3: izz

```
struct VYAListType

 VYApixel    pixel[64];
;

void IZZ::main()
{
 struct VYAListType  Cin, Cout;

 while (true)
 {
   acquireData(CinP);
   load(CinP, 0, Cin);
   releaseRoom(CinP);
    for (unsigned int i=0; i<64; i++)
    {
        Cout.pixel[zigzag[i]] = Cin.pixel[i];
    }
   acquireRoom(CoutP);
   store(CoutP, 0, Cout);
   releaseData(CoutP);
 }
}
```

Evaluation: Compared to vector operation, scalar with array data type

Criteria	Observation	Weight Factors	Scores
1. Number of communication primitives needed	idct1d line 7-9, 32-34; izz line 10-12, 17-19: To send or receive a token, three primitives are needed.	1	1
2. Number of parameters needed in communication primitives	idct1d line 7-9, 32-34; izz line 10-12, 17-19: For multiple token operations, each acquire needs two arguments, each data transfer needs three arguments and each release needs two arguments.	1	2
3. Lines of code	idct1d line 7-9, 32-34; izz line 10-12, 17-19: A clear separation of computation part and communication part leads to more lines of code.	1	1
4. Number of loops	No extra loops are needed.	1	3
5. Complexity of data structures	idct1d line 4, izz line 7: A structure of an array data type is used for multiple token communication.	2	2
6. Code readability	A clear separation of computation and communication parts brings more code readability.	3	3
7. Number of applicable cases	It is not suitable for data dependent communication.	1	1
8. Error-proneness	Change of one part of code can affect the other part due to the change of the relative distance.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d(1) line 3: For each communication session, local memory size of multiple tokens is required.	3	2
Final Score:			42

Table 10.3: Evaluation of Separate, scalar with Array Data Type, In-Order Intra-task Synchronisation and Indirect Access with Relative References

– is difficult to use if the number of tokens needed to be transfer is unknown at compile time.

Summary: The ranking of the scenario is as in Table 10.3:

10.4 Conclusions

The evaluation shows that with indirect access, absolute references is easier to use because of its advantage of less error-proneness over relative references. A vector operation is applicable to more cases than a scalar with array data type operation. Indirect access with absolute references requires less design effort than direct access in general, whereas indirect access with relative references is relatively difficult to use. Thus from the ease of programming point of view, we suggest to support indirect access with absolute references.

So far all the code examples we have showed use blocking synchronisation, which means a synchronisation operation will not return until the communication peer is ready to communicate. In the case of FIFO communication, it means there are enough full tokens for reading or enough rooms for writing. The other way of implementation is to let the synchronisation operations return immediately once it is proven that the synchronisation point is not reached yet. This is called non-blocking synchronisation, which will be discussed in the next chapter.

11 Non-blocking Synchronisation

A non-blocking synchronisation is an operation that returns unsuccessfully when a communication partner has not reached the corresponding synchronisation point. The initial idea is that when non-blocking synchronisation is used, the number of context switches or busy waiting can be reduced if the task has something else to do while waiting. There are options for non-blocking synchronisation operations. One is test and re-acquire and the other one is test and acquire. Each of them can be used in two types of tasks: interrupt service routines and busy waiting tasks. In this chapter, we present four scenarios in four sections, which represent test and reacquire used in interrupt service routine (Section 11.1); test and reacquire with busy waiting (Section 11.2); test and acquire used in interrupt service routine (Section 11.4); test and acquire with busy waiting (Section 11.5).

11.1 Non-blocking, Vector Synchronisation, Test and Re-Acquire Used in Interrupt Service Routines with Relative References

Description: With test and re-acquire, tokens in the channel that have been acquired and not yet released can be acquired again. That means the synchronisation operation starts from the first token that has not been released in the channel. For some systems, it is not too costly do context switches, then the tasks can be implemented as interrupt service routines. After a task returns, the OS will schedule the execution of other tasks. Once the tokens or rooms the task is waiting for are ready in the channel, the task will be fired again.

Suggested Solution: Instead of using a while true loop to keep a task running, an interrupt routine returns every time a data processing is finished. To avoid unnecessary communication and data processing, we first check the availability of all the data and rooms needed if possible. If not possible, not release the early acquired tokens until all the tokens have been proved available.

CASE 1: idct1d

```
void IDCT1D::main()
{
          VYApixel Y[8];
          VYApixel z1[8], z2[8], z3[8];

          /* Check the availability of the data
           and room first. If not available,
            return. */
      if ( !(testReAcquireData(CIn, 8) &&
            testReAcquireRoom(Cout, 8) ) )
        return;

      load(CIn, 0, Y, 8);
      releaseRoom(CIn, 8);

        // Stage 1:
        but(Y[0], Y[4], z1[1], z1[0]);
        z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
        z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
        but(Y[1], Y[7], z1[4], z1[7]);
        z1[5] = CMUL(23170, Y[3]);
        z1[6] = CMUL(23170, Y[5]);

        // Stage 2:
        but(z1[0], z1[3], z2[3], z2[0]);
        but(z1[1], z1[2], z2[2], z2[1]);
        but(z1[4], z1[6], z2[6], z2[4]);
        but(z1[7], z1[5], z2[5], z2[7]);

        // Stage 3:
        z3[0] = z2[0];
        z3[1] = z2[1];
        z3[2] = z2[2];
        z3[3] = z2[3];
        z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
        z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
        z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
        z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

        // Stage 4:
        but(z3[0], z3[7], Y[7], Y[0]);
        but(z3[1], z3[6], Y[6], Y[1]);
        but(z3[2], z3[5], Y[5], Y[2]);
        but(z3[3], z3[4], Y[4], Y[3]);

      store(COut, 0, Y, 8);
      releaseData(COut, 8);
}
```

CASE 2: hs

```
void HS::main()
{
  VYApixel        pixel;
  VYAlineLength   inLineLength;
  VYAlineLength   outLineLength;
  unsigned int        scaleFactor;

  if ( ! ( testReAcquireData(inLineLengthInP)
     && testReAcquireData(outLineLengthInP)))
     /* An extra exit */
     return;
```

```
 load(inLineLengthInP, 0, inLineLength);

 load(outLineLengthInP, 0, outLineLength);

if (!( testReAcquireData(CinP, inLineLength)
    && testReAcquireRoom(CoutP, outLineLength) ) )
 /* An extra exit */
     return;

 // Keep the tokens available in the channel for next testReAquire until
 // now  when it's sure  all the variables needed from communication
 // input  are ready. Otherwise the loaded variables will get lost after the
 // task returns.
 releaseRoom(inLineLengthInP);
 releaseRoom(outLineLengthInP);

 scaleFactor = ceil_div(outLineLength,inLineLength);

 VYApixel* inLine = new VYApixel[inLineLength];
 VYApixel* outLine = new VYApixel[outLineLength];

 for (unsigned int i=0; i<inLineLength; i++)
 {
    load(CinP, i, pixel);
     unsigned int m = i*scaleFactor;
     unsigned int n = std::min(m+scaleFactor,outLineLength);
     for (unsigned int j=m; j<n; j++)
     {
       store(CoutP, j, pixel);
     }
 }
 releaseRoom(CinP, inLineLength);
 releaseData(CoutP, outLineLength);

 delete [] inLine;
 delete [] outLine;
}
```

CASE 3: izz

```
void IZZ::main()
{
 VYApixel    Cin[64];
 VYApixel    Cout[64];

 if ( ! (testReAcquireData(CinP, 64)
 && testReAcquireRoom(CoutP, 64) ) )
    /* Multiple exits exist.*/
    return;

 for (unsigned int i=0; i<64; i++)
 {
    load(CinP, i, Cin);
     store(CoutP, zigzag[i], Cin);
 }
 releaseRoom(CinP, 64);
 releaseData(CoutP, 64);
}
```

CASE 4: filter

```
void Filter::main()
{
  int j, k, m;
  int coeff = 1;

  int* v = new int [npixels];

  m = npixels*nlines;
  if ( ! (testReAcquireData(in,m) && testReAcquireRoom(out, m)) )
     return;

  if (testReAcquireData(cof, npixels)== 1)
  {
     load(cof, 0, coeff);
     releaseRoom(cof);
  }

  for (j=0; j<nlines; j++)
  {
     m = j*nlines;
     load(in, m, v, npixels);
     for (k=0; k<npixels; k++)
     {
       store(out, m+k,  coeff*v[k]);
     }
     releaseRoom(in, npixels);
     releaseData(out, npixels);
  }
  delete [] v;
}
```

Evaluation: Non-blocking Synchronisation is less easy to use than Blocking Synchronisation because

– in most of the multimedia processing cases the task can not proceed with something else if the communication input from a channel (normally the input of the task function) is not ready, which means usually a context switch or busy waiting polling is needed. And they are better to be hidden from the application designer by means of being implemented as blocking synchronisation;

– if interrupt service routines are used, the code is less readable than blocking synchronisation because the task often has multiple exits.

– the code will be confusing if non-blocking tasks are mixed with blocking tasks in a same program.

Summary: The ranking is done as in Table 11.1:

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 5, 7, 8, 31, 32; hs line 7, 9-14, 20, 25, 28, 29; izz line 5, 9, 10, 12, 13, filter 7, 9, 11, 12, 17, 20, 22, 23: To send or receive a token, three primitives are needed.	1	1
2. Number of parameters needed in communication primitives	idct1d line 5, 7, 8, 31, 32; hs line 7, 9-14, 20, 25, 28, 29; izz line 5, 9, 10, 12, 13, filter 7, 9, 11, 12, 17, 20, 22, 23: For multiple token operations, each acquire needs two arguments, each data transfer needs four arguments and each release needs two arguments.	1	1
3. Lines of code	idct1d line 5; hs line 7, 11; izz line 5: With every synchronisation action, two lines of code are needed to deal with the possible unsuccessful result. A clear separation of computation part and communication part leads to more lines of code.	1	1
4. Number of loops	No extra loops are needed.	1	3
5. Complexity of data structures	idct1d line 3, izz line 3, 4; filter 5: With a vector operation, an array data type is used for multiple token communication.	2	2
6. Code readability	A synchronisation operation has to contain a conditional statement and an extra exit, which makes code less readable.	3	1
7. Number of applicable cases	It is suitable for any communication types.	1	3
8. Error-proneness	With relative references, change of one part of code can affect the other part due to the change of the relative distance.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 3, izz line 3, 4; filter 5: With a vector operation, for each multiple token communication session, local memory size of multiple tokens is required.	3	2
Final Score:			37

Table 11.1: Evaluation of Non-blocking, Vector Synchronisation, Test and Re-Acquire Used in Interrupt Service Routines with Relative References

11.2 Non-blocking, Vector Synchronisation, Test and Re-Acquire With Busy Waiting and Relative References

Description: Non-blocking synchronisation can also be used in a busy waiting fashion, where the task keeps polling and waiting for the synchronisation point until it is available. This is useful when the system does not support context switches.

Suggested Solution: Use while loops to keep polling when a non-blocking synchronisation operation returns unsuccessfully.

CASE 1: idct1d

```
1 void IDCT1D::main()
2 {
3           VYApixel Y[8];
4           VYApixel z1[8], z2[8], z3[8];

            /* Loops are needed for polling. */
5         while ( ! testReAcquireData(CIn, 8) );

6         load(CIn, 0, Y, 8);
7         releaseRoom(CIn, 8);

            // Stage 1:
8           but(Y[0], Y[4], z1[1], z1[0]);
9           z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
10          z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
11          but(Y[1], Y[7], z1[4], z1[7]);
12          z1[5] = CMUL(23170, Y[3]);
13          z1[6] = CMUL(23170, Y[5]);

            // Stage 2:
14          but(z1[0], z1[3], z2[3], z2[0]);
15          but(z1[1], z1[2], z2[2], z2[1]);
16          but(z1[4], z1[6], z2[6], z2[4]);
17          but(z1[7], z1[5], z2[5], z2[7]);

            // Stage 3:
18          z3[0] = z2[0];
19          z3[1] = z2[1];
20          z3[2] = z2[2];
21          z3[3] = z2[3];
22          z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
23          z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
24          z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
25          z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

            // Stage 4:
26          but(z3[0], z3[7], Y[7], Y[0]);
27          but(z3[1], z3[6], Y[6], Y[1]);
28          but(z3[2], z3[5], Y[5], Y[2]);
29          but(z3[3], z3[4], Y[4], Y[3]);

30        while( ! testReAcquireRoom(Cout, 8) );

31        store(COut, 0, Y, 8);
32        releaseData(COut, 8);
33 }
```

CASE 2: hs

```
1 void HS::main()
2 {
 VYApixel        pixel;
 VYAlineLength   inLineLength;
 VYAlineLength   outLineLength;
 unsigned int        scaleFactor;

 while ( ! testReAcquireData(inLineLengthInP) );
 load(inLineLengthInP, 0, inLineLength);
 releaseRoom(inLineLengthInP);

 while ( ! testReAcquireData(outLineLengthInP) );
 load(outLineLengthInP, 0, outLineLength);
 releaseRoom(outLineLengthInP);

 while ( ! testReAcquireData(CinP, inLineLength) );
 while ( ! testReAcquireRoom(CoutP, outLineLength) );

 scaleFactor = ceil_div(outLineLength,inLineLength);

 VYApixel* inLine = new VYApixel[inLineLength];
 VYApixel* outLine = new VYApixel[outLineLength];

 for (unsigned int i=0; i<inLineLength; i++)
 {
    load(CinP, i, pixel);
     unsigned int m = i*scaleFactor;
     unsigned int n = std::min(m+scaleFactor,outLineLength);
     for (unsigned int j=m; j<n; j++)
     {
       store(CoutP, j, pixel);
     }
 }
 releaseRoom(CinP, inLineLength);
 releaseData(CoutP, outLineLength);

 delete [] inLine;
 delete [] outLine;
}
```

CASE 3: izz

```
void IZZ::main()
{
 VYApixel    Cin[64];
 VYApixel    Cout[64];

 while ( ! testReAcquireData(CinP, 64) );
 while ( ! testReAcquireRoom(CoutP, 64) );

 for (unsigned int i=0; i<64; i++)
 {
    load(CinP, i, Cin);
    store(CoutP, zigzag[i], Cin);
 }
 releaseRoom(CinP, 64);
 releaseData(CoutP, 64);
}
```

CASE 4: filter

```
void Filter::main()
{
  int j, k;
  int coeff = 1;

  int* v = new int [npixels];

 if (testReAcquireData(cof, npixels)== 1)
 {
    load(cof, 0, coeff);
    releaseRoom(cof);
 }

 for (j=0; j<nlines; j++)
 {
    while( ! testReAcquireData(in, npixels) );
    load(in, 0, v, npixels);

    while ( ! testReAcquireRoom(out, npixels) );
     for (k=0; k<npixels; k++)
     {
       store(out, k,  coeff*v[k]);
    }
    releaseRoom(in, npixels);
    releaseData(out, npixels);
     }
     delete [] v;
}
```

Evaluation: Compared to Non-blocking, Vector Synchronisation, Test and Re-Acquire Used in Interrupt Service Routines, Non-blocking, Vector Synchronisation, Test and Re-Acquire with Busy Waiting is

+ easier to program because the application designer does not need to concern the loss of the local variables after a task returns due to the unavailability of tokens or rooms in latter acquire functions;

+ the code usually has only one exit, which makes it more readable;

– but still a while loop has to be involved;

Summary: We rank the scenario as in Table 11.2:

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 5-7, 30-32; hs line 7-12; izz line 4, 5, 8, 9, 11, 12; filter 6-9, 13-15, 18, 20-21: To send or receive a token, three primitives are needed.	1	1
2. Number of parameters needed in communication primitives	idct1d line 5-7, 30-32; hs line 7-12; izz line 4, 5, 8, 9, 11, 12; filter 6-9, 13-15, 18, 20-21: For multiple token operations, each acquire needs two arguments, each data transfer needs four arguments and each release needs two arguments.	1	1
3. Lines of code	idct1d line 5, 30; hs line 7, 10, 13, 14; izz line 4, 5, filter 6, 13, 15: With relative references, every communication session needs three operations.	1	1
4. Number of loops	idct1d line 5, 30; hs line 7, 10, 13, 14; izz 4, 5, 13, 15: Extra loops are needed for polling.	1	2
5. Complexity of data structures	idct1d line 3, izz line 3, 4; filter 5: With a vector operation, an array data type is used for multiple token communication.	2	2
6. Code readability	Single exit brings more code readability, but a synchronisation still involves a while loop.	3	2
7. Number of applicable cases	It is suitable for any communication types.	1	3
8. Error-proneness	With relative references, change of one part of code can affect the other part due to the change of the relative distance.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 3, izz line 3, 4; filter 5: With relative references, for each multiple token communication session, local memory size of multiple tokens is required.	3	2
Final Score:			39

Table 11.2: Evaluation of Non-blocking, Vector Synchronisation, Test and Re-Acquire with Busy Waiting and Relative References

11.3 Non-blocking, Vector Synchronisation, Test and Acquire Used in Interrupt Service Routines with Relative References

Description: With test and acquire, tokens that have been acquired are no longer available to be acquired. So an acquire operation will start with the first token in the channel that has not yet been acquired.

Suggested Solution: Define private variables in the task class to keep track on which tokens have been acquired already.

CASE 1: idct1d

Here we will define a private variable in class IDCT1D to remember if the data has been acquired or not.

```
...
private:
   /* More memory is required to store the state of acquirement. */
   int dataAcquired = 0;
...

void IDCT1D::main()
{
      VYApixel Y[8];
      VYApixel z1[8], z2[8], z3[8];

   if ( ! dataAcquired )
   {
      if ( ! testAcquireData(CIn, 8) )  return;

   // Data has been acquired.
      dataAcquired = 1;
   }

   if ( ! testAcquireRoom(Cout, 8) ) return;

   // Room has been acquired.
   load(CIn, 0, Y, 8);
   releaseRoom(CIn, 8);

    // Stage 1:
    but(Y[0], Y[4], z1[1], z1[0]);
    z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
    z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
    but(Y[1], Y[7], z1[4], z1[7]);
    z1[5] = CMUL(23170, Y[3]);
    z1[6] = CMUL(23170, Y[5]);

    // Stage 2:
    but(z1[0], z1[3], z2[3], z2[0]);
    but(z1[1], z1[2], z2[2], z2[1]);
    but(z1[4], z1[6], z2[6], z2[4]);
    but(z1[7], z1[5], z2[5], z2[7]);

    // Stage 3:
    z3[0] = z2[0];
    z3[1] = z2[1];
    z3[2] = z2[2];
    z3[3] = z2[3];
    z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
    z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
    z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
    z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

    // Stage 4:
```

```
but(z3[0], z3[7], Y[7], Y[0]);
but(z3[1], z3[6], Y[6], Y[1]);
but(z3[2], z3[5], Y[5], Y[2]);
but(z3[3], z3[4], Y[4], Y[3]);

store(Cout, 0, Y, 8);
releaseData(Cout, 8);

// reset the state variable.
dataAcquired = 0;
}
```

CASE 2: hs

Here we change inLineLength and outLineLength from local variables in the task to private members of the class and initialize them. Besides that, we add a private variable dataAcquired to remember if the inLine data has been acquired or not.

```
...
private:
  /* More memory is required to store the state of acquirement. */
 VYAlineLength   inLineLength = -1;
 VYAlineLength   outLineLength = -1;
 int             dataAcquired = 0;
...

void HS::main()
{
 VYApixel        pixel;
 unsigned int        scaleFactor;

 if (inLineLength == -1)
 {
    if ( ! testAcquireData(inLineLengthInP) ) return;

    load(inLineLengthInP, inLineLength);
    releaseRoom(inLineLengthInP);
 }

 if (outLineLength == -1)
 {
    if( !  testAcquireData(outLineLengthInP) ) return;

    load(outLineLengthInP, outLineLength);
    releaseRoom(outLineLengthInP);
 }

 if ( ! dataAcquired )
 {
    if ( ! testAcquireData(CinP, inLineLength) ) return;
    dataAcquired = 1;
 }

 if ( ! testAcquireRoom(CoutP, outLineLength) ) return;

 scaleFactor = ceil_div(outLineLength,inLineLength);
for (unsigned int i=0; i<inLineLength; i++)
{
    load(CinP, i, pixel);
    unsigned int m = i*scaleFactor;
    unsigned int n = std::min(m+scaleFactor,outLineLength);
    for (unsigned int j=m; j<n; j++)
    {
      store(CoutP, j, pixel);
    }
}
releaseRoom(CinP, inLineLength);
releaseData(CoutP, outLineLength);

delete [] inLine;
delete [] outLine;

// reset the state variables.
inLineLength = -1;
outLineLength = -1;
dataAcquired = 0;
}
```

CASE 3: izz

Like in case 1, we add a variable dataAcquired as a private member to the class.

```
...
private:
  /* More memory is required to store the state of acquirement. */
  int dataAcquied = 0;
...
```

```
void IZZ::main()
{
  VYApixel    Cin[64];
  VYApixel    Cout[64];

  if ( ! dataAcquired )
  {
     if ( ! testAcquireData(CinP, 64) ) return;

     dataAcquired = 1;
  }

  if ( ! testAcquireRoom(CoutP, 64) ) return;

  for (unsigned int i=0; i<64; i++)
  {
     load(CinP, i, Cin);
     store(CoutP, zigzag[i], Cin);
  }
  releaseRoom(CinP, 64);
  releaseData(CoutP, 64);

  // reset the state variable.
dataAcquired = 0;
}
```

Evaluation: Both used in interrupt service routines , compared to Test-and-Re-Acquire, Test-and-Acquire

– needs more design effort to deal with the state of the acquirements.

– has less code readability because there are too many algorithms for communication behaviour involved, which are mixed with the computation behaviour of the tasks.

Summary: We rank it as in Table 11.3

11.4 Non-blocking, Vector Synchronisation, Test and Acquire With Busy Waiting with Relative References

Description: In a busy waiting approach, the task will keep polling until the tokens it acquires are available.

Suggested Solution: Not like in the previous scenario, the state of data acquirements is not necessary to be remembered.

CASE 1: idct1d

```
void IDCT1D::main()
{
        VYApixel Y[8];
        VYApixel z1[8], z2[8], z3[8];

        /* Loops are needed for polling. */
      while ( ! testReAcquireData(CIn, 8) );

      load(CIn, 0, Y, 8);
      releaseRoom(CIn, 8);

        // Stage 1:
        but(Y[0], Y[4], z1[1], z1[0]);
        z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
        z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
        but(Y[1], Y[7], z1[4], z1[7]);
        z1[5] = CMUL(23170, Y[3]);
        z1[6] = CMUL(23170, Y[5]);

        // Stage 2:
        but(z1[0], z1[3], z2[3], z2[0]);
        but(z1[1], z1[2], z2[2], z2[1]);
        but(z1[4], z1[6], z2[6], z2[4]);
        but(z1[7], z1[5], z2[5], z2[7]);

        // Stage 3:
        z3[0] = z2[0];
```

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 9, 12-14, 37, 39; hs line 11-13, 17-19, 23, 26, 30, 35, 38-39; izz line 7, 10, 13, 14, 16, 17, 18: To send or receive a token, three primitives are needed.	1	1
2. Number of parameters needed in communication primitives	idct1d line 9, 12-14, 37, 39; hs line 11-13, 17-19, 23, 26, 30, 35, 38-39; izz line 7, 10, 13, 14, 16, 17, 18: For multiple token operations, each acquire needs two arguments, each data transfer needs four arguments and each release needs two arguments.	1	1
3. Lines of code	idct1d line 2, 7, 10, 39; hs line 2-4, 9, 15, 21, 24, 42-44; izz line 2, 5, 8, 18: Extra lines of code have to be spent to deal with the state of data acquirement.	1	1
4. Number of loops	No extra loops are needed.	1	3
5. Complexity of data structures	idct1d line 5, izz line 3, 4: With a vector operation, an array data type is used for multiple token communication.	2	2
6. Code readability	The code used to deal with the state of data acquirement makes the separation of communication and computation parts seems less clear. With relative reference, the task has to keep track on the relative distance of the current token to the oldest one that has been acquired but not yet released. Sometimes it is not the same as the index of the current token in the token stream it acquires, which leads to less code readability.	3	1
7. Number of applicable cases	It is suitable for any communication types.	1	3
8. Error-proneness	With relative references, change of one part of code can affect the other part due to the change of the relative distance.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 2, 5; hs 2-4; izz line 2, 3, 4: Extra memory space has to be spent on holding the state of the acquirement. With relative references, for each multiple token communication session, local memory size of multiple tokens is required. Besides extra memory is required to hold the state of data acquirement.	3	1
Final Score:			34

Table 11.3: Evaluation of Non-blocking, Vector Synchronisation, Test and Acquire Used in Interrupt Service Routine with Relative References

```
        z3[1] = z2[1];
        z3[2] = z2[2];
        z3[3] = z2[3];
        z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
        z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
        z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
        z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

        // Stage 4:
        but(z3[0], z3[7], Y[7], Y[0]);
        but(z3[1], z3[6], Y[6], Y[1]);
        but(z3[2], z3[5], Y[5], Y[2]);
        but(z3[3], z3[4], Y[4], Y[3]);

      while( ! testReAcquireRoom(Cout, 8) );

      store(COut, 0, Y, 8);
      releaseData(COut, 8);
}
```

CASE 2: hs

```
void HS::main()
{
  VYApixel        pixel;
  VYAlineLength   inLineLength;
  VYAlineLength   outLineLength;
  unsigned int        scaleFactor;

  while ( ! testReAcquireData(inLineLengthInP) );
  load(inLineLengthInP, 0, inLineLength);
  releaseRoom(inLineLengthInP);

  while ( ! testReAcquireData(outLineLengthInP) );
  load(outLineLengthInP, 0, outLineLength);
  releaseRoom(outLineLengthInP);

  while ( ! testReAcquireData(CinP, inLineLength) );
  while ( ! testReAcquireRoom(CoutP, outLineLength) );

  scaleFactor = ceil_div(outLineLength,inLineLength);

  VYApixel* inLine = new VYApixel[inLineLength];
  VYApixel* outLine = new VYApixel[outLineLength];

  for (unsigned int i=0; i<inLineLength; i++)
  {
     load(CinP, i, pixel);
      unsigned int m = i*scaleFactor;
      unsigned int n = std::min(m+scaleFactor,outLineLength);
      for (unsigned int j=m; j<n; j++)
      {
        store(CoutP, j, pixel);
      }
  }
  releaseRoom(CinP, inLineLength);
  releaseData(CoutP, outLineLength);

  delete [] inLine;
  delete [] outLine;
}
```

CASE 3: izz

```
void IZZ::main()
{
  VYApixel    Cin[64];
  VYApixel    Cout[64];

  while ( ! testReAcquireData(CinP, 64) );
  while ( ! testReAcquireRoom(CoutP, 64) );

  for (unsigned int i=0; i<64; i++)
  {
     load(CinP, i, Cin);
     store(CoutP, zigzag[i], Cin);
  }
  releaseRoom(CinP, 64);
  releaseData(CoutP, 64);
}
```

CASE 4: filter

```
void Filter::main()
{
  int j, k;
  int coeff = 1;

  int* v = new int [npixels];

  if (testReAcquireData(cof, npixels)== 1)
  {
     load(cof, 0, coeff);
     releaseRoom(cof);
  }

  for (j=0; j<nlines; j++)
  {
     while( ! testReAcquireData(in, npixels) );
     load(in, 0, v, npixels);

     while ( ! testReAcquireRoom(out, npixels) );
      for (k=0; k<npixels; k++)
      {
        store(out, k,  coeff*v[k]);
     }
     releaseRoom(in, npixels);
     releaseData(out, npixels);
      }
      delete [] v;
}
```

Evaluation: Compared to Non-blocking, Vector Synchronisation, Test and Re-Acquire Used in Interrupt Service Routines, Non-blocking, Vector Synchronisation, Test and Re-Acquire with Busy Waiting is

+ easier to program because the application designer does not need to concern the loss of the local variables after a task returns due to the unavailability of tokens or rooms in latter acquire functions;

+ the code usually has only one exit, which makes it more readable;

– but still a while loop has to be involved;

Summary: We rank the scenario as in Table 11.4:

11.5 Conclusions

The assessment shows that in the approach of interrupt service routine, test and re-acquire is easier to use than test and acquire. Busy waiting is easier than with interrupt service routines. Blocking synchronisation is better than non-blocking synchronisation because no effort is required to handle the unsuccessful synchronisation result and the code with blocking synchronisation is easier to read. Thus we suggest to support blocking synchronisation instead of non-blocking synchronisation. But if Non-blocking Synchronisation is supported, we suggest to support test and re-acquire instead of test and acquire.

Obviously it is handy if the TTL infrastructure knows about the computational semantics of data types, like in all the code examples we have discussed so far. However, sometimes, to support only simple communication data types, e.g. bytes or bits is desired. What will the application code will like then? In the next chapter, we will discuss low-level communication data types.

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 5-7, 30-32; hs line 7-12; izz line 4, 5, 8, 9, 11, 12; filter 6-9, 13-15, 18, 20-21: To send or receive a token, three primitives are needed.	1	1
2. Number of parameters needed in communication primitives	idct1d line 5-7, 30-32; hs line 7-12; izz line 4, 5, 8, 9, 11, 12; filter 6-9, 13-15, 18, 20-21: For multiple token operations, each acquire needs two arguments, each data transfer needs four arguments and each release needs two arguments.	1	1
3. Lines of code	idct1d line 5, 30; hs line 7, 10, 13, 14; izz line 4, 5, filter 6, 13, 15: With relative references, every communication session needs three operations.	1	1
4. Number of loops	idct1d line 5, 30; hs line 7, 10, 13, 14; izz 4, 5, 13, 15: Extra loops are needed for polling.	1	2
5. Complexity of data structures	idct1d line 3, izz line 3, 4; filter 5: With a vector operation, an array data type is used for multiple token communication.	2	2
6. Code readability	Single exit brings more code readability, but a synchronisation still involves a while loop.	3	2
7. Number of applicable cases	It is suitable for any communication types.	1	3
8. Error-proneness	With relative references, change of one part of code can affect the other part due to the change of the relative distance.	3	2
9. Code reusability	It has no specific requirements or different solutions to different platforms, which means more code reusability.	3	3
10. Memory usage	idct1d line 3, izz line 3, 4; filter 5: With relative references, for each multiple token communication session, local memory size of multiple tokens is required.	3	2
Final Score:			39

Table 11.4: Evaluation of Non-blocking, Vector Synchronisation, Test and Acquire with Busy Waiting and Relative References

Low-level Communication Data Types 12

Until now, all the scenarios above use high-level communication data types, which requires the infrastructure to have the knowledge about the semantics of data types. Sometimes, in order to more accurately model the use of the communication resources or have a simpler infrastructure implementation, the system designers may wish the tasks to send information from one to another in the form of (multiple) bits or bytes (called low-level communication data types), without the infrastructure knowing the semantics of data types. In this chapter, we discuss low-level communication data types.

12.1 Low-level Communication Data Types, Combined, Synchronisation and Data Transfer

Depending on whether the task is designed with the same endianness assumption, there are two situations existing:

- The producer and the consumer use the same endianness
- The producer and the consumer use different endianness.

The code examples of the two situations will be presented below. Because it is not up to TTL to choose a situation out of the two, the evaluation will be based on the worst case: different endianness.

12.1.1 Same Endianness

Description: This is the scenario in which the producer and the consumer use the same endianness, no matter whether it is the big endianness or the small endianness.

Suggested Solution: Convert the high-level data types to low-level data types when tokens are to transferred through channels.

CASE 1: idct1d

```
void IDCT1D::main()
{
        VYApixel Y[8];
        VYApixel z1[8], z2[8], z3[8];

        while (true)
        {
```

```
      /* Convert high-level data types to low-level data types. */
        read(CIn, (unsigned char *)Y, sizeof(VYApixel)*8);

            // Stage 1:
            but(Y[0], Y[4], z1[1], z1[0]);
            z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
            z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
            but(Y[1], Y[7], z1[4], z1[7]);
            z1[5] = CMUL(23170, Y[3]);
            z1[6] = CMUL(23170, Y[5]);

            // Stage 2:
            but(z1[0], z1[3], z2[3], z2[0]);
            but(z1[1], z1[2], z2[2], z2[1]);
            but(z1[4], z1[6], z2[6], z2[4]);
            but(z1[7], z1[5], z2[5], z2[7]);

            // Stage 3:
            z3[0] = z2[0];
            z3[1] = z2[1];
            z3[2] = z2[2];
            z3[3] = z2[3];
            z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
            z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
            z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
            z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

            // Stage 4:
            but(z3[0], z3[7], Y[7], Y[0]);
            but(z3[1], z3[6], Y[6], Y[1]);
            but(z3[2], z3[5], Y[5], Y[2]);
            but(z3[3], z3[4], Y[4], Y[3]);

        /* Convert high-level data types to low level data types. */
        write(COut, (unsigned char *)Y, sizeof(VYApixel)*8);
    }
}
```

CASE 2: hs

```
void HS::main()
{
    while (true)
    {
        VYApixel    pixel;
        VYAlineLength   inLineLength;
        VYAlineLength   outLineLength;
        unsigned int    scaleFactor;

      read(inLineLengthInP, (unsigned char *)&inLineLength,
         sizeof(VYAlineLength));
      read(outLineLengthInP, (unsigned char *)&outLineLength,
         sizeof(VYAlineLength));

        scaleFactor = ceil_div(outLineLength,inLineLength);
        VYApixel* inLine = new VYApixel[inLineLength];
        VYApixel* outLine = new VYApixel[outLineLength];

        read(CinP, (unsigned char *)inLine, sizeof(VYApixel)*inLineLength);
        for (unsigned int i=0; i<inLineLength; i++)
        {
            unsigned int m = i*scaleFactor;
            unsigned int n = min(m+scaleFactor,outLineLength);
            for (unsigned int j=m; j<n; j++)
            {
                outLine[j] = inLine[i];
            }
        }
      write(CoutP, (unsigned char *)outLine,
         sizeof(VYApixel)*outLineLength);

        delete [] inLine;
        delete [] outLine;
    }
}
```

CASE 3: izz

```
void IZZ::main()
{
    VYApixel    Cin[64];
    VYApixel    Cout[64];

    while (true)
    {
      read(CinP, (unsigned char *)Cin,
            sizeof(VYApixel)*64);
        for (unsigned int i=0; i<64; i++)
        {
            Cout[zigzag[i]] = Cin[i];
        }
      write(CoutP, (unsigned char *)Cout,
            sizeof(VYApixel)*64);
    }
}
```

12.1.2 Different Endianness

Description: This is the scenario in which the producer and the consumer use different endianness, no matter big endianness or small endianness.

Suggested Solution: Convert the high-level data types to low-level data types when tokens are to transferred through channels. Besides that, on either the producer side or the consumer side, the endianness of data has to be shifted.

CASE 1: idct1d

```
template < class T >
void EndianessShift(T* t)
{
 T temp;

 /* An extra loop is needed. */
 for (int i=0; i¡sizeof(T)/2; i++)
 {
     temp = ((unsigned char *)t)[sizeof(T)-1-i];
     ((unsigned char *)t)[sizeof(T)-1-i] = ((unsigned char *)t)[i];
     ((unsigned char *)t)[i] = temp;
 }
}

void IDCT1D::main()
{
     VYApixel Y[8];
     VYApixel z1[8], z2[8], z3[8];

     while (true)
     {
       read(CIn, (unsigned char *)Y, sizeof(VYApixel)*8);
      /* An extra loop is needed. */
       for (int i=0; i¡8; i++)
       {
          EndianessShift(&Y[i]);
       }

        // Stage 1:
        but(Y[0], Y[4], z1[1], z1[0]);
        z1[2] = SUB(CMUL( 8867, Y[2]), CMUL(21407, Y[6]));
        z1[3] = ADD(CMUL(21407, Y[2]), CMUL( 8867, Y[6]));
        but(Y[1], Y[7], z1[4], z1[7]);
        z1[5] = CMUL(23170, Y[3]);
        z1[6] = CMUL(23170, Y[5]);

        // Stage 2:
        but(z1[0], z1[3], z2[3], z2[0]);
        but(z1[1], z1[2], z2[2], z2[1]);
        but(z1[4], z1[6], z2[6], z2[4]);
        but(z1[7], z1[5], z2[5], z2[7]);

        // Stage 3:
        z3[0] = z2[0];
        z3[1] = z2[1];
        z3[2] = z2[2];
        z3[3] = z2[3];
        z3[4] = SUB(CMUL(13623, z2[4]), CMUL( 9102, z2[7]));
        z3[7] = ADD(CMUL( 9102, z2[4]), CMUL(13623, z2[7]));
        z3[5] = SUB(CMUL(16069, z2[5]), CMUL( 3196, z2[6]));
        z3[6] = ADD(CMUL( 3196, z2[5]), CMUL(16069, z2[6]));

        // Stage 4:
        but(z3[0], z3[7], Y[7], Y[0]);
        but(z3[1], z3[6], Y[6], Y[1]);
        but(z3[2], z3[5], Y[5], Y[2]);
        but(z3[3], z3[4], Y[4], Y[3]);

      write(COut, (unsigned char *)Y, sizeof(VYApixel)*8);
    }
}
```

CASE 2: hs

```
template< class T >
void EndianessShift(T* t)
{
  T temp;

/* An extra loop is needed. */
  for (int i=0; i¡sizeof(T)/2; i++)
  {
     temp = ((unsigned char *)t)[sizeof(T)-1-i];
     ((unsigned char *)t)[sizeof(T)-1-i] = ((unsigned char *)t)[i];
     ((unsigned char *)t)[i] = temp;
  }
}
```

```
void HS::main()
{
  while (true)
  {
      VYApixel    pixel;
      VYAlineLength   inLineLength;
      VYAlineLength   outLineLength;
      unsigned int    scaleFactor;

     read(inLineLengthInP, (unsigned char *)&inLineLength,
        sizeof(VYAlineLength));
     EndianessShift(&inLineLength);
     read(outLineLengthInP, (unsigned char *)&outLineLength,
        sizeof(VYAlineLength));
     EndianessShift(&outLineLength);

      scaleFactor = ceil_div(outLineLength,inLineLength);

      VYApixel* inLine = new VYApixel[inLineLength];
      VYApixel* outLine = new VYApixel[outLineLength];

     read(CinP, (unsigned char *)inLine,
        sizeof(VYApixel)*inLineLength);

      /* An extra loop is needed. */
      for (int i=0; i<inLineLength; i++)
      {
        EndianessShift(&inLine[i]);
      }
      for (unsigned int i=0; i<inLineLength; i++)
      {
          unsigned int m = i*scaleFactor;
        unsigned int n = min(m+scaleFactor,outLineLength);
        for (unsigned int j=m; j<n; j++)
        {
            outLine[j] = inLine[i];
        }
    }
    write(CoutP, (unsigned char *)outLine,
         sizeof(VYApixel)*outLineLength);

    delete [] inLine;
    delete [] outLine;
 }
}
```

CASE 3: izz

```
template< class T >
void EndianessShift(T* t)
{
  T temp;

  /* An extra loop is needed. */
  for (int i=0; i¡sizeof(T)/2; i++)
  {
     temp = ((unsigned char *)t)[sizeof(T)-1-i];
     ((unsigned char *)t)[sizeof(T)-1-i] = ((unsigned char *)t)[i];
     ((unsigned char *)t)[i] = temp;
  }
}

void IZZ::main()
{
 VYApixel    Cin[64];
 VYApixel    Cout[64];

 while (true)
 {
    read(CinP, (unsigned char *)Cin, sizeof(VYApixel)*64);
  /* An extra loop is needed. */
    for (int i=0; i¡64; i++)
    {
       EndianessShift(&Cin[i]);
    }
    for (unsigned int i=0; i<64; i++)
    {
        Cout[zigzag[i]] = Cin[i];
    }
    write(CoutP, Cout, 64);
 }
}
```

Evaluation: As we can see from the examples above, low-level communication data type is more difficult to program:

– because all the high level data types that are easy to use for the application designer must be converted to bytes to make it suitable for communication;

– because the endianness can be different for different architectures, to leave the problem to the application designer is not good for the reuse of the application components;

Summary: We can rank it as in Table 12.1

12.2 Conclusions

From the comparison, we see that low-level communication data types are very difficult to use because the tasks have to deal with the endianness if the producer and the consumer use different endianness. We suggest to support high-level communication data types only. Having presented all the scenarios we listed in Chapter 7, we will summarise and conclude the thesis in the next chapter.

Criteria	**Observation**	**Weight Factors**	**Scores**
1. Number of communication primitives needed	idct1d line 18, 45; hs line 9, 11, 16, 30; izz line 18, 27: With combined synchronisation and data transfer, to send or receive a token, only one argument is needed.	1	3
2. Number of parameters needed in communication primitives	idct1d line 18, 45; hs line 9, 11, 16, 30; izz line 18, 27: Each of read and write needs only three primitives.	1	3
3. Lines of code	idct1d line 1-11, 19-22; hs line 1-11, 10, 12, 17-20; izz line 1-11, 19-22: Many lines of code are used to handle the different endianness.	1	1
4. Number of loops	idct1d line 5-10, 19-22; hs line 5-10, 17-20; izz line 5-10, 19-22: Extra loops are added to handle the different endianness.	1	1
5. Complexity of data structures	idct1d line 18, 45; hs line 9, 11, 16, 30; izz line 18, 27: Conversion to byte pointer type is needed, which is rather complex.	2	1
6. Code readability	idct1d line 1-11, 18-22, 45; hs line 1-11, 10-12, 16-20; izz line 1-11, 18-22, 27: The code dealing with the endianness is confused with the computation part of the code. Data type conversions also lower the code readability.	3	1
7. Number of applicable cases	It is suitable for any communication behaviour type: data dependent or independent.	1	3
8. Error-proneness	Pointers are used. Besides the change of the communication peer may lead to the change of endianness difference between the producer and the task.	3	1
9. Code reusability	Reuse the task with different communication peer may lead to the change of endianness difference between the producer and the task, which further causes the change of the code.	3	1
10. Memory usage	idct1d line 14; hs line 14-15; izz line 14-15: For each multiple token communication session, local memory size of multiple tokens is required.	3	2
Final Score:			28

Table 12.1: Low-level Communication Data Types, Combined, Synchronisation and Data Transfer

13 Summary and Conclusions

The increasing demands on complex systems-on-chip design call for a change in design methodology for heterogeneous embedded systems.

In this thesis, we applied the Y-chart system design method, which allows the design of the application and the architecture of a system to be decoupled, the components of the two parts to be re-targeted to other systems and performance analysis to be done at different abstract levels.

In Chapter 2, we described a standardized platform interface called the Task Transaction Level(TTL), which supports the use of Y-chart system method. We showed that the design of the TTL interface needs to take not only the implementation's performance in clock cycles into account but also the application designers' effort in using the TTL interface.

In Chapter 3, we defined that our objectives were to evaluate the TTL inter-task communication interfaces in the aspect ease of programming from application designers' point of view. To achieve that, in Chapter 5, we proposed a seven-step approach to assess the design effort needed from each design choice in a quantitative way.

Following the approach, in Chapter 6-7, the characteristics of the inter-task communication behaviour of multi-media processing applications were studied; a set of benchmark code was selected according to the study; a group of criteria were proposed and their corresponding weight factors were defined in a ordinal scale; the TTL design choices were investigated and based on all that, we proposed a comparison scheme which defined the scenarios we used to compare against each other.

In subsequent chapters(8-12), we applied the quantitative approach to several scenarios. For each scenario, by assessing the code examples of according to the group of criteria and calculating the weighted sum, we obtained a final score. These final scores are given in Table 13.1 and can be summarized as follows:

After this series of comparison, we conclude that from the application designer point of view, in the ease of program aspect,

- combined synchronisation and data transfer is better than separate synchronisation and data transfer;
- vector token data transfer is easier than scalar when there are multiple tokens to be transferred at one time;
- when we use separate synchronisation and data transfer, indirect access is easier than direct access;
- when indirect access is used, with absolute token references is easier than with relative token references;

Scenarios	**Final Scores**
Combined Scalar Synchronisation and Data Transfer	50
Combined Scalar with Array Data Type Synchronisation and Data Transfer	40
Combined Vector Synchronisation and Data Transfer	51
Separate, Vector, In-Order Intra-task Synchronisation and Direct Access	41
Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Direct Access	43
Separate Vector, Out-of-Order Intra-task Synchronisation and Direct Access	43
Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Absolute References	47
Separate, Vector, In-Order Intra-task Synchronisation and Indirect Access with Relative References	43
Separate, Scalar with Array Data Type, In-Order Intra-task Synchronisation and Indirect Access with Relative References	42
Non-blocking, Vector Synchronisation, Test and Re-acquire Used in Interrupt Service Routines with Relative References	37
Non-blocking, Vector Synchronisation, Test and Re-acquire Used with Busy Waiting and Relative References	39
Non-blocking, Vector Synchronisation, Test and Acquire Used in Interrupt Service Routines with Relative References	34
Non-blocking, Vector Synchronisation, Test and Acquire Used with Busy Waiting and Relative References	39
Low-level Communication Data Types, Combined, Synchronisation and Data Transfer	39

Table 13.1: Summary of evaluation

- blocking synchronisation is easier than non-blocking synchronisation;
- when we use non-blocking synchronisation, test and re-acquire is easier than test and acquire;
- high-level communication data type is more convenient than low-level communication data type.

These conclusions clearly show the comparison of the design effort required from different design choices in the TTL inter-task communication. These conclusions together with the results of an implementation's performance in clock cycle study will give the TTL designers a clear picture of which advantages and disadvantages each design choice has and thus can be directly adopted in the TTL inter-task communication specification making. The TTL application code in this thesis can be used by TTL application designers as examples. Furthermore, the quantitative approach that is proposed in this thesis can be applied to the application design effort study for other embedded system interface design as it is able to steer the design process in its early phase and it offers prior information on the behaviour of certain components.

Bibliography

[1] `http://ptolemy.eecs.berkeley.edu`.

[2] André Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Rafael Peset Llopis, Ramanathan Sethuraman, Natalino Bus, and Kees Goossens, *C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems*, Design Automation for Embedded Systems (2002).

[3] Andrew S. Tanenbaum, *Computer Networks*, Prentice Hall, 1989.

[4] Andy D. Pimentel, Louis O. Hertzberger, Paul Lieverse, Pieter van der Wolf, Ed F. Deprettere, *Exploring embedded-systems architectures with artemis*, IEEE (2001), 57–62.

[5] Daniel C. Hyde, *Introduction to programming language occam*, `http://www.eg.bucknell.edu`.

[6] E.A. de Kock and G. Essink, *Y-chart application programmer's interface, the yapi programmer's and reference guide version 0.5*, Nat.Lab. Technical Note 2001/144 (2001).

[7] W.J.M Smits P. van der Wolf J.-Y. Brunel W.M. Kruijtzer P. Lieverse E.A. de Kock, G. Essink and K.A. Vissers, *Yapi: Application modeling for signal processing systems*, Proceedings 37th Design Automation Conference (DAC) (2000), 402–405.

[8] Fiddler, J.; Wilner, D.N.; Wong, H., *Multiprocessing: an extension of distributed, real-time computing*, "Compcon Spring '90. 'Intellectual Leverage'. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.", Feb-Mar "1990".

[9] George Fankhauser, Burkhard Stiller, Bernhard Plattner, *Arrow: A Flexible Architecture for an Accounting and Charging Infrastructure in the Next Generation Internet*, `www.tik.ee.ethz.ch/ gfa/papers/netnomics-fin.pdf`.

[10] Gerben Essink, Andrei Radulescu, Pieter van der Wolf, Jeffrey Kang, *Task transaction layer*, under preparation.

[11] Hoare, C.A, R., *Communications Sequential Processes*, Prentice Hall, 1985.

[12] Integrated System Inc., *pSOSystem System Concepts*, 1995.

[13] John L. Hennessy, David A. Patterson, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann Publishers, 1996.

[14] G. Kahn, *The semantics of a simple language for parallel programming*, Proc. of Information Processing (1974), 471–475.

[15] B. Kienhuis, *An approach for quantitative analysis of application-specific dataflow architectures*, Proc. Int'l Conf. Application-Specific Systems, Architectures, and Processors, IEEE CS Press, Los Alamitos, Calif. (1997), 338–349.

[16] Magnus Jonssen, *Real-time Communication*, `http://www.hh.se/staff/magnusj /rt-com/`.

[17] Neugass, H.; Espin, G.; Nunoe, H.; Thomas, R.; Wilner, D., *VxWorks: an interactive development environment and real-time kernel for Gmicro*, TRON Project Symposium, 1991. Proceedings., Eighth (1991), 196 –207.

[18] Om Prakash Gangwal, André Nieuwland, Paul Lippens, *A scalable and flexible data synchronisation scheme for embedded hw sw shared memory systems*, Proceedings of the international symposium on Systems synthesis **14** (2001).

[19] Rajesh Gupta, *Task and Task Management*, `http://www.cs.ucsd.edu/ gupta/ cse291-s03/Lecture9TasksOS.pdf`.

A
An Example of a YAPI Program

The main program (main.cc):

```
#include "pc.h"     // declaration file for the process network.
#include "yapi.h"       // declaration file for YAPI

int main() {
    RTE rte;        // a YAPI class derived from runtime environment

    // create the process network
    PC pc( id("pc") );

    // start the process network
    rte.start(pc);

    return 0;
}
```

The process network declaration (pc.h)

```
#include "yapi.h"
#include "producer.h"   // declaration file for the producer process
#include "consumer.h"   // declaration file for the consumer process

class PC :  public ProcessNetwork
// ProcessNetwork is a YAPI defined base class for user defined process netwrok
{ public:
    PC(const Id& n);
    const char* type() const;
private:
    Fifo<int> fifo;         // Fifo is a YAPI defined class for fifos.

    Producer producer;      // class Producer is defined in "producer.h"
    Consumer consumer;      // class Consumer is defined in "consumer.h"
};

Process Network Implementation (pc.cc)
#include "pc.h"

PC::PC (const Id& n) :
    ProcessNetwork( n ),
    fifo( id ("fifo") ),
    producer( id( "producer" ), fifo ),
    consumer( id( "consumer" ), fifo )
{}

const char* PC::type() const {
    return "PC";
}
```

The Producer Process Declaration (producer.h)

```
#include "yapi.h"

class Producer : public Process
// Process is a YAPI defined class for user-defined process
{ public:
    producer (const Id& n, Out<int>& o);
    const char* type() const;
    void main();

private:
    OutPort<int> out;
};
```

Producer Process Implementation (producer.cc)

```
#include "producer.h"
#include <iostream>

Producer::Producer(const Id& n, Out<int>& o) :
    Process(n),
    out( id( "out" ), o )
{}

const char* Producer::type() const {
    return "Producer";
} void Producer::main() {
    std::cout << "Producer started" << std.endl;

    const int n = 1000;

    write( out, n );

    for (int i=0; i<n; i++)
    {
        write( out, i );
}

std::out << type() << " " << fullName() << ": " << n << " values
written" <<std::endl; }
```

Consumer Process Declaration (consumer.h)

```
#include "yapi.h"
class Consumer : public Process { public:
    Consumer( const Id& n, In<int>& I);
    const char* type() const;
    void main();

private:
    InPort<int> in;
};
```

Consumer Process Implementation (consumer.cc)

```
#include "consumer.h"
#include <assert.h>
#include <iostream.h>

Consumer::Consumer( const Id& n, In<int>& I) :
    Process( n ),
    in ( id ("in"), i )
{}

const char* Consumer::type() const {
    return "Consumer";
}

void Consumer::main() {
    int n, j;

    std::cout << "Consumer started" << std::endl;

    read( in, n );

    for (int i=0; i<n; i++)
{
    read( in, j );
    assert( i=j );
}

std::cout << type() << " " <<fullName() << ": " << n << " values
read" << std::endl;

}
```

B Inter-Task Communication Decision Points

Here we list a series of issues that affect the primitives in TTL inter-task communication.

- Reliable vs. Unreliable Channels
- Ordered vs. Unordered Channels
- In-order vs. Out-of-order Production and Consumption of Data
- Uni-directional vs. Bi-directional Channels
- Low-Level vs. High-Level Communication Data Types
- Single-Type vs. Multi-Type Channels
- Zero, One or Multiple Consumers
- Single vs. Multiple Producers
- Blocking vs. Non-Blocking Synchronisation
- Communication via Shared Variables
- Combined vs. Separated Synchronisation and Data Transfer
- Destructive vs. Non-destructive Write
- Single vs. Multiple Tokens Available for Data Transfer
- In-Order vs. Out-of-Order Data Token Transfers
- In-Order vs. Out-of-Order Inter-Task Synchronisation
- In-Order vs. Out-of-Order Intra-Task Synchronisation
- Local vs. Non-local Tokens
- Direct vs. Indirect Access
- Absolute vs. Relative Token References
- Single vs. Multiple Token Operations
- Blocking vs. Non-Blocking Read, Write and Acquire
- Test Only vs. Test and Acquire
- Full vs. Partial Read, Write and Acquire

Curriculum Vitae

Yanning Luo was born in Shanghai, China on the 21st of October 1974. In 1993, she was admitted into Shanghai Jiao Tong University. She received her bachelor degree in the faculty of Electronics Engineering at Shanghai Jiao Tong University in July 1997. From 1997 to 1999, she worked as a programmer in Inventec (Shanghai) Electronic Co., Ltd. From 1999 to 2001, she worked as a software engineer in Lucent Technologies of Shanghai.

In September 2001, she started her MSc study in Electrical Engineering at Delft University of Technology (TU Delft), The Netherlands. In November 2002, she started working on her MSc thesis at Philips Research Laboratories (Nat.Lab.), Eindhoven under the supervision of Pieter van der Wolf, Gerben Essink (Philips Nat.Lab.) and Koen Bertels (Computer Engineering Lab - CE, TU Delft). The CE Laboratory is part of the Department of Electrical Engineering, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, and is chaired by Prof. Stamatis Vassiliadis. Her Msc thesis is entitled: "TTL Interface for Multiprocessor Platforms". Her research interests include: Embedded Systems, Telecommunication and Computer Architecture.