MONASH
AUSTRALIA'S INTERNATIONAL UNIVERSITY

Faculty of Computing and Information Technology

Department of Robotics and Digital Technology

Technical Report 94-6

Caroline Network Control Program Interface
Version 1.0

Rik Harris
Anthony McGregor
Cameron Blackwood

February 8, 1995

Enquiries:-

Technical Report Coordinator
Robotics and Digital Technology
Monash University
Clayton VIC 3168
Australia

tr.coord@rdt.monash.edu.au +61 3 905 3402

Contents

Abstract

This paper describes the Network Control Program Interface (NCPI), the interface between the Caroline Control Software, and the Caroline Operating Software. Operating Software includes the Caroline Simulator and the Caroline Network Node. The interface was defined in such a way as to allow different versions of the Control Software and Operating Software, as long as they follow the interface specification. So, an implementation of the routing and control code could be used on both a simulator and a Caroline node, or several different types of Caroline node. It allows new control software to be written without requiring the modification of the software in the nodes' firmware. The interface could also be used to create, for example, automated testing programs to test nodes or versions of the control software.

Keywords

- Caroline
- Network
- Cell Switching
- ATM
- Asynchronous Transfer Mode

1 Introduction

1.1 Definitions

Most terms used in the Caroline project are defined in [2], however this paper introduces several terms, which are defined here:

Control Software
: The software which provides the routing and control functions for a Caroline node. All nodes in a network must use compatible Control Software, as all instances of the Control Software in a network cooperate to provide the network service.

Operating Software
: The software which provides the abstraction of the hardware, to allow nodes with different construction to run identical Control Software. The Operating Software is usually stored in a node's firmware. Most of the functionality of the Operating Software is defined in this document.

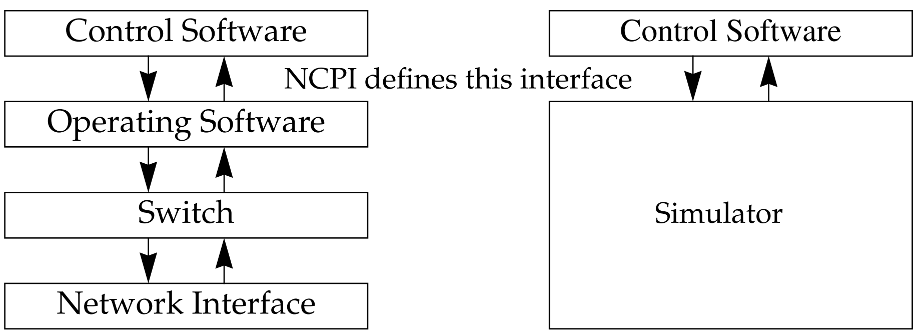
1.2 Scope

It is expected that this document will be used by two groups of people. The first group are the writers of simulators of the Caroline network and the writers of the node's Operating Software, which will usually be installed on every node in firmware. The second group are the protocol developers who implement the Control Software which provides routing, naming, and all other tasks which are not provided by the Caroline hardware or firmware. Before reading this paper, you should be familiar with the Caroline Network project and aims [2], the Network layer interface [3], the Caroline hardware [4], and the C programming language [8]. The Control Software which has been implemented at this time uses a flooding algorithm to determine paths through the network [1].

Version 1.0 of the Network Control Program Interface definition (NCPI) specifies the interface as is implemented by version 1.x of the Control Software, and version 0.8 of the Caroline simulator. It is expected that a revised version of this interface definition will be released at a later date.

1.3 Structure of Caroline

The Caroline node and simulator can be logically broken down into several layers



1.3.1 Network Interface Layer

The Network Interface layer is comprised of up to seven Input Output Units. Each unit is used to communicate with another node in a Caroline Network. The Network Interface layer is responsible for:

- transferring cells between nodes, and
- any buffering and internode flow control which may be required for the internode links.

For a node to communicate with an adjacent node, the Network Interface layer module must implement the same physical internode communication protocol on each Input Output Unit as its corresponding adjacent node's Input Output Unit.

1.3.2 Switch Layer

The Switch layer is contains the CPU, Address Resolution Unit, etc (see [4] for more details). The Switch layer is responsible for:

- forwarding data cells between the Input Output Units in the Network Interface layer once a virtual circuit has been established,
- collecting statistics,
- forwarding control cells from the Operating Software layer to the appropriate module in the Network Interface layer,
- providing some mechanims for implementing timers (such as clock interrupts, or a real-time clock), and
- forwarding incoming control cells to the Operating Software layer.

As long as the interfaces definitions are adhered to, the Switch layer can be different on each node in a Caroline network. A particular Switch layer implementation usually corresponds with an appropriate Operating Software layer.

1.3.3 Operating Software Layer

The Operating Software provides an abstraction of the node hardware (Switch layer) which allows a single implementation of the Control Software to run on different types of Caroline node. The Operating Software layer is responsible for:

- implementing the interface described in this document,
- loading and initialising the Control Software,
- setting the contents of the Address Resolution Unit (ARU), when requested by the Control Software,
- keeping track of multiple timers and notifying the Control Software when a timer expires (using the timing mechanism in the Switch layer), and
- passing cells between the Switch layer and the Control Software layer.

As long as the interfaces definitions are adhered to, the Operating Software layer can be different on each node in a Caroline network. A particular Operating Software layer is usually only used for a corresponding Switch layer. There is no particular reason why the Operating Software layer must be kept separate from the Switch layer, except for convenience. The Operating Software layer is usually installed as firmware in a node's firmware.

1.3.4 Control Software Layer

The Control Software implements the protocols which allow circuits to be established across a Caroline network. The Control Software layer is responsible for:

- Implementing the routing protocol to establish virtual circuits,
- determining the contents of the Address Resolution Unit (ARU), to allow the Switch layer to switch data cells to their appropriate destination, and passing that information to the Operating Software layer, and
- any other (non-data) inter-node communication which is required.

For a node to participate in a Caroline network, the Control Software must implement the same protocol as all other nodes in the network. The Control Software uses the interface described in this document.

1.4 Layer Interfaces

For the Caroline network to operate with a variety of types of nodes interconnecting with many types of communications media, the interfaces between each of these layers must be well-defined. These interfaces have been documented in three papers:

1. The interface between the Network Interface layer and the Switch layer is described in [3].
2. A formal definition of the interface between the Operating Software and Switch layer is not as important as the definitions of the other interfaces. Each node with a different Switch layer will have an appropriate, corresponding Operating Software layer. An example interface between the node Switch and Operating Software layers is described in [4].
3. This document describes the interface between the Operating Software and Control Software layers.

1.5 Return Values

In most cases, functions are defined as returning an `integer` 1 on success, and an `integer` 0 on failure. These values correspond to the C language's TRUE and FALSE respectively, so the following constructs can be used:

```
if (get_cpu_cell(buffer))
{
  process cell
}
else
{
  handle failure
}
```

which makes the programmer's intentions more obvious.

2 Operating Software

The following describes the Operating Software side of the interface. The functions described in this section should be implemented by the Operating Software. All functions must be implemented, even if they perform no action. The Control Software uses these functions to implement its functionality. The Control Software must not use any other mechanisms to access the node hardware. Operating Software functions are non-reentrant, although interrupt routines may need to be reentrant. Table 2.1 summarises the functions that must be implemented by the Operating Software.

FIGURE 2.1. Summary of Operating Software Functions

Function	Description
`set_aru(in_port, in_vc, out_port, out_vc)`	Set the contents of the ARU
`get_cpu_cell(buffer)`	Copy a cell from the CPU input queue to the buffer
`put_cpu_cell(buffer)`	Copy a cell from the buffer to the CPU output queue
`get_time()`	Return the value of the node's clock
`set_time(val)`	Set the node's clock
`add_timer(val)`	Set a timer to expire when the node's clock reaches *val* ms
`del_timer(val)`	Unset a previously set timer
`feedback(msg)`	Display a feedback message
`debug(msg)`	Display a debugging message
`get_nodename()`	Return the node's assigned name

2.1 Address Resolution Unit Manipulation

The Control Software must have complete control over the contents of the ARU. It is permissible for the Operating Software to initialise the ARU when the node is initialised, but the Control Software should not depend on the ARU being in any particular state when it starts. After any part of the Control Software has been executed, however, the Operating Software is not to modify the ARU, except where the `set_aru()` function is called:

```
int set_aru(int in_port, int in_vc, int out_port,
     int out_vc)
```

Set the ARU location for `in_port`, `in_vc` to `out_port`, `out_vc`. This means that all cells arriving at the node on port `in_port` with the virtual circuit identifier of `in_vc` will be directed by hardware to port out_port and have their virtual circuit identifier translated to out_vc. The set_aru function returns 1 when it has successfully set the ARU location, and 0 if it was unable to perform the operation. The most likely reason for failure is that one of the parameters was outside the appropriate range.

As no other layer is allowed to modify the ARU tables, the Control Software can expect that the table contents will not change, so it can safely store a shadow table of ARU contents. There is no mechanism for the Control Software to read the ARU tables once they have been set (as manipulations of the ARU are slow compared to RAM accesses), so maintaining a shadow table of the ARU contents is the only way to access the ARU data once the set_aru() function has been called.

2.2 Cell Handling

In the current implementation, Caroline cells are 16 bytes long and may contain any data, (values from 0-255), so the buffer is not null terminated. The buffer used by all cell handling functions must have at least 16 bytes of storage. This space must be allocated by the Control Software.

```
int get_cpu_cell(char *buffer)
```

The get_cpu_cell() call is used by the Control Software to transfer a cell from the CPU input queue to a buffer that the Control Software can access. The get_cpu_cell() function returns a 1 if a cell was available, and the buffer was filled, and a 0 if no cell was available.

```
int put_cpu_cell(char *buffer)
```

The put_cpu_cell() call is used by the Control Software to store a cell in the CPU output queue, to be forwarded to one of the node's output ports. The cell is transferred from the CPU output queue through the node, including being subjected to an ARU translation to choose which output port the cell should be sent on. The put_cpu_cell() function returns a 1 if the cell was successfully passed to the Switch layer, and a 0 if the Operating Software was unable to send the cell. The Operating Software should silently drop the cell if it is destined for a port which has nothing connected to it, and should return a 0 if the contents are otherwise invalid, such as the CPU output queue being full.

The basic flooding algorithm [5] [7] shows how the ARU translations can be used to determine where a cell came from, and how to send a cell on a specific port.

2.3 Timer Handling

```
long get_time()
```

Return the value of the internal node real-time clock (in milliseconds). If the clock has not been set by set_time(), then it returns number of milliseconds since the node was initialised.

`long set_time(long val)`

Set the internal node real-time clock. If the time is set forward, timers that were set to expire between the time before calling `set_timer()` and the new time should be expired, in order, by calling `process_exipry()` for each timer, when control next returns to the Operating Software. The Operating Software must not call `process_expiry()` before the `set_timer()` call returns, as the Control Software is not reentrant. If `process_expiry()` must be called multiple times, to remove all timers, the Operating Software must wait for the previous `process_expiry()` to return before calling it again (i.e. interrupt handlers should not call the `process_expiry()` function). If the time is set backward, timers which have expired in the past must not trigger another call to `process_exipiry()`.

`void add_timer(long val)`

Add a timer to the internal timer list. The time `val` is the time measured in milliseconds from the same reference as the node clock. The Operating Software layer should call the Control Software function `process_expiry()` (see section 3.2) at the next available opportunity after that time is reached. For example, to request that `process_expiry()` is called in 20 milliseconds time, the Control Software would execute something like: `add_timer(get_time()+20);`

`void del_timer(long val)`

Remove a timer from the internal timer list. If there are multiple timers set for this time, only one should be removed.

The Operating Software must call the `process_expiry()` function after a timer expires. However, it must not call `process_expiry()` until after the Control Software has returned execution control to the Operating Software. It is up to the Control Software to keep track of which timers have been set, and to act accordingly. The basic flooding code [7] contains an example of how the Control Software could manipulate many timer events.

2.4 Program Feedback

It is expected that most protocol development work will be carried out using the simulator. This allows for debugging and informational messages to be displayed, so the developer can determine what the code is doing. It seems likely, however, that the developer may also want some form of feedback while the software is executing on the real node hardware, as well. To support this, two functions must be provided by the node interface. They are:

`void feedback (char *msg, ...)`

Display an informational message regarding control software operation. The feedback messages are expected to be of use to the network manager of an operational network.

`void debug (char *msg, ...)`

Display a message for debugging purposes. The debugging messages are only likely to be of use to the protocol developer.

Both of these functions use Variable-length Argument Lists, as described in [8]. An example implementation of the `feedback()` function is shown:

```
void   feedback(msg)
char   *msg;
{
  va_list    va;

  va_start(va, msg);
  vprintf(msg, va);
  va_end(va);
}
```

It is expected that a mechanism will be provided to suppress the output of each of these functions. It is not unreasonable for the Operating Software layer of a node with no means of data output to implement these functions as null operations.

2.5 Miscellaneous

Each node should have a unique identifier associated with the hardware:

```
char *get_nodename()
```

The `get_nodename` function returns a null pointer if the node has no name associated with it, or a null-terminated string, if a name is available. This means that software using version 1.0 of the NCPI will not be able to use node names with null characters in them.

2.6 Code Structure

The Operating Software contains the main execution loop, and calls functions in the Control Software where appropriate. The basic structure of the Operating Software could be:

```
init_control()
while(1)
{
        wait_for_event()
        if (cell available) process_cell();
        if (timer expired) process_expiry(time);
}
```

If the `wait_for_event()` function is not present, the Operating Software will be polling. It may be appropriate to implement a blocking mechanism in `wait_for_event()`, but that is up to the implementors of the Operating Software. It should have no impact on the operation of the Control Software.

3 Control Software

This section describes the Control Software side of the interface. The functions described in this section should be implemented by the Control Software. All functions must be implemented, even if they perform no action. Once the Operating Software has called one of these functions, it must not call another until the first has finished. This means that the interrupt handlers in the Operating Software must buffer the interrupts until the Control Software has returned control (by `returning` from these functions). This allows the developer to write non-reentrant code. Table 3.1 summarises the functions that must be implemented by the Control Software.

FIGURE 3.1. Summary of Control Software Functions

Function	Description
`init_control()`	Initialise the Control Software
`process_expiry(val)`	Process a timer expiry event
`process_cell()`	Process an incoming control cell

When the Control Software is to be tested on the simulator it will be compiled and linked with the simulator code. The simulator will call the `init_control()` function when it starts, and the `process_expiry()` and `process_cell()` functions when a timer expires, or a cell arrives, respectively. An interface module implements the Operating Software functions described in chapter 2 (see [6] for more specific information about using the Caroline Simulator).

When the Control Software is compiled for the hardware it will be compiled and linked with the hardware versions of the functions described in chapter 2. The Operating Software in the node will call the `init_control()` when the node is initialised, `process_expiry()` when a timer expires, and `process_cell()` when a cell arrives.

3.1 Initialisation

When the node is initialised, the Operating Software will call the following function in the Control Software:

```
void init_control()
```

This function should initialise the ARU, and any variables required by the Control Software. The `init_control()` function should be called by the Operating Software only at node initialisation time.

3.2 Event Handling

The Control Software should provide the following functions:

```
void process_expiry(long val)
```

This function is called by the Operating Software when a timer, added by `add_timer()` expires. The time that the timer was set for is passed in the `val` parameter. It is up to the Control Software to determine *which* timer has expired, based on this time. The prototype Control Software contains code to keep track of an arbitrary number of timers.

```
void process_cell()
```

This function is called by the Operating Software when a cell, destined for the node's CPU, is received. It is up to the Control Software to call `get_cpu_cell()` to actually retrieve the cell.

3.3 Code Structure

The basic structure of the node Control Software could be something like the program on the following page.

```
void init_control()
{
        initialise aru
        initialise other variables
}


void process_expiry(val)
{
        int          timer;
        timer=find_timer(val); /* find timer which expired */
        if (timer == a)
        {
              process timer a
        }
        if (timer == b)
        .
        .
        .
}


void process_cell()
{
        char         buffer[16], buffer2[16];
        static long  future;

        get_cpu_cell(buffer);
        port=buffer[0] & 0x1f;
        type=buffer[1];
        if (type=1)
        {
              process cell type 1
              put_cpu_cell(buffer2);
              add_timer(future=(get_tick()+200));
        }
        if (type=2)
        {
              process cell type 2
              del_timer(future);
        }
        if (type == 3)
        .
        .
        .
}
```

A more complete example can be found in the first version flooding code, in [5], and a description of the implementation of the basic flooding algorithm can be found in [7]. Information on an implementation of the Caroline Simulator can be found in [6].

References

[1] Breen, J. W. (1994) "The Caroline Project: Service and Protocol Description." Monash University, Department of Robotics and Digital Technology, Technical Report 93-15.

[2] Blackwood, C., Harris, R., and McGregor, A. J. "Introduction to the Caroline Network." Monash University, Department of Robotics and Digital Technology, not yet published.

[3] Blackwood, C., Harris, R. and McGregor, A. J. (1994) "Caroline Node Interface Specification and Implementor's Guide." Monash University, Department of Robotics and Digital Technology, Technical Report 94-5.

[4] Blackwood, C. and Harris, R. "Caroline Node Hardware Specification and Design Guide." Monash University, Department of Robotics and Digital Technology, not yet published.

[5] Harris, R. (1995) "Scalable Routing in a Packet Switched Network." MComp Thesis. Monash University, Department of Robotics and Digital Technology, not yet published.

[6] Harris, R. (1994) "Installation and Use of the Caroline Simulator." Monash University, Department of Robotics and Digital Technology, Technical Report 94-7.

[7] Harris, R. "Implementation of the Caroline Basic Flooding Algorithm." Monash University, Department of Robotics and Digital Technology, not yet published.

[8] Kernighan, B., Ritchie, D. M. (1988) "The C Programming Language" Prentice Hall. Second Edition.