RTI interface between PC and DSP

Doug Rogers
(Edited by Peter Hancock)

Version 5	:	30^{th}	April	2002
Version 4	:	7^{th}	March	2002
Version 3	:	15^{th}	February	2002
Version 2	:	29^{th}	January	2002
Version 1	:	6^{th}	January	2002

Contents

Version information, changes

From version 1 to version 2, the following changes were made:

- DSP handler kill processing was altered.
- Sections were added for:
  - PC handler index
  - PC handler
  - PC port array
  - PC device list
- The PC device section was simplified.

Handler removal is now completed in one call with the handler and ports dying back automatically. The handler states have been much reduced. The matchmaker states have now been incorporated into the PC port state and the PC device state[1]

From version 1 to version 2, the following changes were made:

- The document was ported from Wordperfect to LaTeX. The grammar was touched up here and there, in the interests of clarity.
- Certain parts of the content were revised to reflect a new, sketch version of the core software ('Hakit') produced by Doug. It does not reflect any functioning stable version of the code. Some parts are entirely speculative, and some are undoubtedly sheer nonsense. The intention is to make a start on producing some documentation of some help to introduce someone unfamiliar with the system to its design.

Apart from formatting, sections pertaining to the DSP were left as they were.

From version 2 to version 3, the following changes were made:

- (Forgotten, I'm afraid.)

From version 3 to version 4, the following changes were made:

- Doug's new state tables added for pc port, dsp port, and pc device object.

From version 4 to version 5, the following changes were made:

- (To be filled in – lots of stuff to be deleted.)

[1] I lifted the previous three sentences from version 2 of the document, where they occurred under the heading 'Differences from DSPOS state machines'. They may or may not say the same as the preceding list, which was produced by comparing the text of the two versions.

1 High level description

In general terms, ED's software consists of a number of components and tools for configuring custom ED installations (wave makers/tanks, wave power generators, ..).

The customer's interface to the system is a gui control-panel, configured system-by-system using a couple of tools: select-and-plug and arrange-layout.

Hardware architecture. There are three levels of processor:

PC, host Runs the front end, gui or control panel. Co-ordinates 'nodes' in a graph prepared using the 'select and plug' tool. The software is written in C++. It uses Microsoft libraries for graphical display, access to the file system, etc.

DSP, hub Manages communication with control processors (aka devices, FPGAs), tracks and controls their state (u/s, running happily, etc). Written in C++. Uses a ED-written kernel/scheduler.

There are actually two versions of this device[2]. They do not have quite the same interface.

FPGA, device, control processor Executes a short downloaded control program, written using a proprietary assembly language, that reads input signals coming from sensors (position, velocity, etc) and writes things to actuators.

Communications.

- The low-level communication interface between the PC and the DSP is the Link mechanism, that works by exchanging blocks/stacks of messages with a buffer in the DSP. (As mentioned above, the low-level mechanism depends on the version of the DSP.)
- The low-level interface between the DSP and the FPGAs is the Disco protocol; this is a synchronous message passing protocol designed by ED for real-time control.

Software architecture. The software is thought of as divided into a 'core', present in every system, and other software which is more concerned with gui presentation, file system access, logging, remote communications (as from a wave-power generator to the shore), and so on.

The core of the system is thought of as a number of communicating state machines, known collectively as 'the state machine'. Some of these run on the PC, and some on the DSP. The interface to the core of the system is represented by a single RTI object, residing on the PC. The chief components of the RTI object are:

[2]Variously known as 'Blacktip' = dsp21k32, and 'Sharc' = dsp21ksf

- a DeviceList (section 10 on page 23), which is a list of Device objects (section 9 on page 20). This object resides only on the PC.
- a PortArray (section 8 on page 17), which is an array of Port objects (section 7 on page 14). All the port objects are built when the DSP and PC code is initialised. They correspond to the individual physical devices (FPGAs) used in a particular installation. Port objects track the state of the physical devices.
- a HandlerIndex (section 12 on page 26), which is an array of Handler indices (section 11 on page 24). as well as message queues (for data and commands), and low-level link data structures. A handler index is a pointer to a handler object, together with a state code.

  Each handler on the PC has a array of device objects (section 11 on page 24, allocated to it when the handler is built. Each device can tolerate a range of identifiers for physical devices. The ranges tolerated by different devices are disjoint (or else the system-image has been misconfigured).

On the DSP side, there are the following major components:

- a handler ID array, reflecting the PC's HandlerIndex. Each DSP handler (section 5 on page 12) has an array of sockets (section 6 on page 13, corresponding to the array of devices associated with a PC handler.
- a port array. A description of a DSP port is given in section 4 on page 9.

Much of the design of the system is focussed on the problem of connecting devices with ports, bring them online and into communication with the system, and removing them from communication when they disappear. This process begins with receipt of a DISCO message received by the DSP via a particular socket, giving the identifier of the FPGA device which is plugged into the physical socket on the DSP. This is communicated to the PC. The initial matching of device and port is effected first on the PC. Then a build message is sent to the DSP, allowing the DSP's handler socket and port on the DSP to link up.

The following remarks occur in version 2 of the document, and seem to be important. I put them here for want of anywhere better.

Order of events for handler insertion into RTI

1. Lock handler ID array
2. Obtain an array entry and use its value as the handler ID.
3. Send build message to handler on DSP with build parameters $(sockets, type, datawidth, ID, flag)$.
4. Set handler state to live (MUST be second to guarantee build message is first always)

5. Release handler array lock
6. Configure the DSP handler
7. Create set of devices in handler array
8. Insert devices into device list (this automatically makes match check)

Order of events for removal of handler from RTI

1. Remove devices from device table (Must be first to prevent match during shutdown)
2. Kill any ports matched to devices, delete device objects
3. Send kill message to DSP handler
4. Reset handler array value (no need to take lock)
5. Set handler state to dead

2 General remarks

State machines All state machine in the following sections are Mealy machines. That is to say: each state and input event determines an output action and next state; there is a distinguished initial state.

In the descriptions of state machine below:

- The states are partitioned into groups, identified by state-codes: for example *Active*, *Owned*, *Dead*. Each state is identified by its state-code, together with the values of certain state-variables, depending on the state-code; for example *StoreID*, *timeout_count*.

- The input events are partitioned into groups, identified by input-codes: for example *IDevent*. Each input is identified by an input-code, together with certain arguments that may depend on the input-code. The input events can also be qualified by a (boolean) *guard*, that may depend on the state, input-code and arguments: for example $timeout\_count > 0$.

  In this document, events seem to fall into one of two classes: procedure calls (which is to say the event of passing control to a procedure[3]), and receipt of a message (which is to say the event of passing control to a procedure that handles messages of a certain form).

- The output actions are similar to input events. They are however simpler, in that there is nothing corresponding to a guard on the output side. Instead of entry to a procedure with input values (arguments), actions model returns from a procedure with result values. Instead of message receipt, actions model messages being sent.

[3]I am not sure whether this is true in all cases: it may be that in some cases execution of an entire procedure constitutes an event.

- The format of the descriptions is something like this:

  States

  variables var_0
  var_1
  ...

  codes $code_0$
  $code_1$
  ...

  Stimulus

  - inbound procedure call or message received. May be guarded by a predicate. If P is a predicate $\overline{P}$ is its negation.
  - ...

  Response

  - outbound procedure call or message sent
  - ...

  Transitions sorted by origin of transition

  - state-code$_0$
  - state-code$_1$
  - ...

- All input events not explicitly mentioned result in an error action. In this version of the document, this has been left vague. The right thing to do probably depends on the particular state machine.

Although the core software is referred to as 'the state machine', this is a slightly poetic figure of speech. In fact there are many communicating state machines. Moreover, it seems (having tried to force everything to fit it) that the model of Mealy machines isn't appropriate in the case of every component. In these cases, a less structured form of description will be used. Still, on the assumption that the structure of components is right, a component will in every case have a state-space spanned by a small number of state variables, an initial state (that may depend on initialisation parameters), and undergo transitions from state to state that depend on commands or input events.

3 Remarks about errors

In my opinion (PGH), it is worth distinguishing between different categories of 'error'.

- bugs. There will always be bugs. They are mistakes in programming. Programming is figuring out code, and that means reasoning. If you do not have a clear statement of what it is that a piece of program is supposed to accomplish, at least in your head, you are not programming, but just writing code.

Without going overboard about it, to have a solid basis for avoiding, detecting, and dealing with bugs, it is essential that there are clear statements somewhere in the vicinity of a given piece of code that say what it is supposed to do, and what it is supposed not to do. If anyone other than the author, or a clairvoyant is to have a solid basis for understanding the code, jargon terms should be avoided, or at least clearly defined in a sacred document.

There are two ways to make such statements: in comments (where one can be as vague as one considers safe), and in assertions which can in principle be executed (where one may have to write a lot of extra code to fully express preconditions, postconditions and invariants as boolean values). The second kind can be very valuable in debugging.

It is worth distinguishing *assumptions* which are required to be established by the calling code from *assertions* which are to be ensured by the called code.

As bugs are inevitable, you should devote any brain power left from trying to avoid them, to devising a minimally embarrasing and maximally informative way of dedecting, reporting and/or recording them. My experience for what it is worth is that a full-scale symbolic dump of major data-structures, together with a trace-buffer is what one wants, *provided* both have been fully exercised, tuned in the light of experience, and ascertained to be completely bomb-proof.

- big-time, catastrophic bad things. The computer isn't working right. There are memory errors, electrical noise fouling something up, spurious interrupts, missed interrupts, power loss, water in the wires, and so on and so on. You have to decide which of these things you are going to try to detect, report, and/or (unlikely) recover from.

- small-time, tiresome bad things. Misconfigurations. User errors, like clashing device identifiers. Stuff that one anticipates happening, and needs to detect, report, and (preferably) recover from, if only in 'cripple-mode'.

- not-errors, but merely rare conditions. It may be that most of the time, you expect to find something on a list. Exceptionally, it will not be there. Perhaps for performance reasons, or perhaps to unclutter the code, you'd like to employ some special mechanism such as throwing and catching exceptions, rather than regular code structures.

- hangs.

Then there are mechanisms that one may employ for dealing with errors, whether they are bugs, catastrophes, annoyances, or not errors at all. I am speaking about exceptions. Exceptions are not a policy for dealing with errors, they are a mechanism that one may use for implementing a policy.

An example of a policy is to try to arrange things so that errors can be easily attributed to specific components, and those components shut down, and possibly reinitialised and restarted.

Exceptions are non-local jumps, and they are intrinsically hard to understand, which is reflected by the fact that it is difficult to write down (correct) rules for reasoning about code which uses exception passing. I don't know much about the form of exception passing used in C++, but I know enough to advocate that one confines one's interest in language features to a very vanilla, unexciting subset of what is available. Preferably, exceptions should be handled only at 'the top level'.

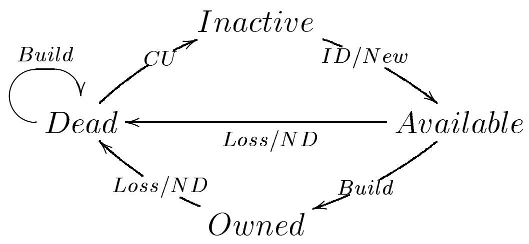
The use of exceptions is complicated by threading, as exceptions are propagated up (down?) a thread's stack, which may (for example) mean that some other mechanism may to be employed to communicate 'an exception' to some other thread which is in a position to handle it. If any data structures have to be locked to raise and/or handle an exception, the risk of deadlock has to be carefully considered.

4 DSP Port

There is an array of ports, that corresponds to the physical ports on the DSP, except that there is one special port –currently port[0] – that provides the time function, and in some sense represents 'the DSP itself'. Each port adds dynamic storage to provide swinging buffers for (1) data transfer in and out (ie. from and to the FPGA). (2) outgoing register read and write packets, (3) an event buffer (introduced as of the first version of this document).

States Inactive
Available
Owned
Dead

State graph



Start State	*End State*	*stimulus*	*response (to pc-port)*
Inactive	Available	Id packet	NewId
Available	Owned	Build from pc-port	
Available	Dead	Loss ID or kill	NowDead
Owned	Dead	Loss ID or kill	NowDead
Dead	Inactive	Cleanup from pc-port	
Dead	Dead	Build from pc-port	

States

variables (integers)
handler
socket
StoreID
timeout_count

codes
Inactive (initial)
Available
Owned
Dead

Build data *handler*
socket
inBuffSize
outBuffSize
regBuffSize
eventBuffSize

WARNING: the content of this section below has not been changed since version 2 of the document.

Inputs

- *IDevent*(*ID*): receipt of an *ID* event from a physical socket. The argument is (one hopes...) the *ID* value physically configured on the FPGA device.
- *Process*(*time*): procedure called once per time slice: 32 times per second at present. I don't know what the argument represents.
- *MesgBuild*(*data*): message from the PC to set up the buffers for communication with the FPGA.
- *MesgCleanup*(): message from the PC to release dynamic storage.
- *Kill*(): procedure called internally or from the handler to which the port is linked.

Outputs

- *handler* :: *Attach* procedure call to socket on handler (as defined by build data)
- *handler* :: *Unattach* procedure call to socket on handler to break attachment
- *MesgNewID*(*ID*) Message sent to the PC port on new ID
- *MesgDead*() Message sent to the PC port on device disconect

Transitions Other state/event combinations than those below are erroneous.

- *Inactive*
  - On *IDevent*(*id*) ,
    send *NewID*(*id*) to PC port,
    set *StoreID* to ID,
    set *timeout_count* to 3,
    next state: *Available*.
- *Available*
  - On receiving *Build*(*data*) from PC,
    $handler :: Attach(handler, socket, port)$,
    Build(*data*),
    next state : *Owned*.

- On $IDevent(ID)$ when $ID = StoreID$,
  set $timeout\_count$ to 3.
  state unchanged.
- On $IDevent(ID)$when $ID \neq StoreID$),
  send $MesgWrongID(ID)$, (* Only a warning! *)
  state unchanged.
- On Process(),
  when $time\_count \neq 0$,
  decrement $time\_count$ by 1,
  state otherwise unchanged.
- On $time\_count = 0$,
  send $NowDead$,
  new state: $Dead$.
- On Kill,
  send $NowDead$,
  set state to $Dead$.

* *Owned*
  - On IDevent(ID), when $ID = StoreID$,
    set $timeout\_count$ to 3,
    state otherwise unchanged.
  - On $IDevent(ID)$, when $ID \neq StoreID$,
    send $MesgWrongID(ID)$, (* Only a warning! *)
    state unchanged.
  - On $Process()$,
    when $time\_count \neq 0$
    then decrement $timeout\_count$ by 1.
    state otherwise unchanged.
  - On $Kill$,
    $handler :: UnAttach()$
    send $NowDead$
    New state: $Dead$
  - When $time\_count = 0$
    call $handler :: UnAttach()$,
    send $NowDead$,
    New state: $Dead$
* *Dead*
  - On $PcMesg(Cleanup)$
    New state: $Inactive$
  - On $PcMesg(build)$
    no change.

5 DSP Handler

WARNING: no changes (beyond formatting) since version 2 of the document.

The DSP handler has no state change except existence. Therefore state can be through a NULL instead of the handler pointer. Configuration of the handler and attachment of all the ports is handled separately as it does not affect the rest of the state machine.

States Array of sockets each with their own status.

Build data Build information

- handler type
- sockets
- kill flag

Inputs Kill actions

- Call to each attached port with kill (this causes the unattach)
- Remove entry from index
- Send free handler index message to PC

Outputs Functions called by the port for a socket and by the socket state machine:

- Attach(socket, port) : Passed to the relevant socket
- Unattach(socket) : See below - then passed to relevant socket

Transitions (Missing.)

If the kill flag is set, then on receiving an unattach message all the attached ports are sent a kill message, and a message is sent to the handler.

6 DSP Socket

States

variables *port* (integer)

codes *Unlinked* (Initial)
Linked

Inputs (Not given.)

Outputs (Not given.)

Transitions

- Unlinked
  - On $handler :: Attach(handler, socket, port)$
    set $storePort$ to $port$
    Next state-code: *Linked*
- Linked
  - On $handler :: Unattach(handler, socket)$
    Next state-code: *Unlinked*

Note:- on unlink if handler has auto suicide then kill called for all linked ports

State machine - actions when not in specified states are fatal errors.

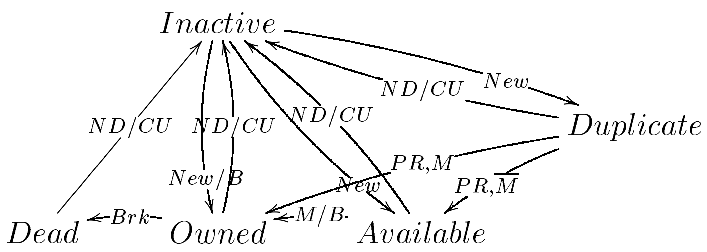
Usage of handler socket to port link: when the link is in place, the port will send all events other than the ID event and DRET event to the attached handler socket.

The data input and output queue are accessed from the handler, thus allowing secure data transfer between the DSP handler and the attached devices.

The event queue (new as of version 1?) will allow the handler to send event packets to a device. This permits near real time data transfer between devices.

7 PC Port

State graph



Start State	*End State*	*stimulus*	*response*
Inactive	Duplicate	NewId, port match	
Inactive	Available	NewId, no device match	
Inactive	Owned	NewId, device match	Build
Duplicate	Available	Port release, no device match	
Duplicate	Owned	Port release, device match	Build
Duplicate	Inactive	NowDead	Cleanup
Available	Owned	device match	Build
Available	Inactive	NowDead	Cleanup
Owned	Dead	Device match break	
Owned	Inactive	NowDead	Cleanup
Dead	Inactive	NowDead	Cleanup

WARNING: The following needs revision, and maybe incorporation into the state graph. Perhaps what remains should be incorporated into the description of the *PortArray*. The functionality described in versions 1 and 2 of this document is more complex than the version of the code on which version 3 is based. All state transitions of a port are handled by calling a *SetState* procedure. This is an 'arrow' based description of the state-graph. Each arrow is given a separate name.

REMARK: When defining state-tables, it is good to sort them first by the source state, then by the destination state.

An arrow is written $\xrightarrow{s/r}$ where the stimulus may be a message or procedure call. It may be guarded by a predicate. The response may be to send a message, or do nothing (except change state).

States

codes *Inactive* (Initial)
Available
Duplicate
Owned
Dead

variables *DevicePtr*
Id

Inputs The following are arguments to *SetState*

- *ToActive*(*id*) – *id* a device identifier
- *ToMatch*(*dev*) – *dev* a device pointer
- *ToDead*
- *ToDuplicate*
- *ToClean*

Outputs None – unless an exception should be described as an output.

Transitions State machine - actions when not in specified states raise exceptions.

- *Inactive* - we get here from anywhere by a *NowDead* from DSP port.
  - *NewId*(*id*),
    when no port match, no device match
    set *Id* to id, *dev* unchanged
    Next code: *Available*
  - *NewID*(*id*),
    when port match,
    set *Id* to id, *dev* unchanged
    Next code: *Duplicate*
  - *NewId*(*i*),
    when device match for i,
    set *Id* to id, *dev* unchanged
    send Build to DSP port
    Next code: *Owned*
- *Duplicate*
  - when port release and no device match
    set *DevPtr* to *dev*
    Next state-code: *Available*
  - when port release and device match ,
    set *DevPtr* to *dev*
    send Build to DSP port
    Next state-code: *Owned*
  - *NowDead*
    trash *DevPtr*
    send Cleanup to DSP port
    Next state-code: *Inactive*

- *Available*
  - when device match,
    set *DevPtr* to *dev*
    send Build to DSP port
    Next state-code: *Owned*
  - *NowDead*
    when $dev \neq NULL$,
    trash *DevPtr*
    Next state-code: *Inactive*
- *Owned*
  - device match break,
    Id, *dev* unchanged
    Next state-code: *Dead*
  - *NowDead*,
    Id, *dev* unchanged
    Next state-code: *Inactive*
- *Dead*
  - *NowDead*,
    Id, *dev* unchanged
    Next state-code: *Inactive*

8 PC PortArray

There is on the PC an array of ports that mirrors the ports on the DSP, and ultimately the real ports of the hub. All operations on a port go through the PortArray interface.

Terminology

identifier An identifier has 3 parts: version number, type code and switch setting.

device identifier matching A device may be matched by a range of version numbers. In essence a device will tolerate a range of identifiers.

similarity of identifiers Two identifiers are similar if they have the same type code and switch setting. (The version does not matter.)

valid identifier An identifier is valid if it is the identifier of an active port.

active port A port is active if it is not inactive. Alternatively, if it is one of the states Duplicate, Available, Owned or Dead.

non-duplicate A port is non-duplicate if it is active, but not Duplicate.

Invariants

1. If we partition the identifiers associated with non-inactive ports into groups with similar identifiers (having the same type and switch number), then in any group there is exactly one port which is non-duplicate (it is necessarily Available, Owned, or Dead).

2. If p is an active port, and there is no active device which matches it in ID, the device is either available or dead. (ie. nether duplicate nor owned.)

3. If d is an active device, and all active ports that match it in ID are duplicate, d's state is necessarily Available.

4. p is a non-duplicate port (ie available, owned or dead), and d is an active device which matches it in ID, then EITHER p's state is owned, and d's state is configuring or active OR p's state is dead, and d's state is available.

   Note this implies that p's state is not available; ie that we don't have states in which p is available and there's some device which matches it in ID.

Other assertions

1. A port is owned only if there is a current device which matches it in ID.
2. If an active port does not match any current device, the port state is non-duplicate.

WARNING: The description below is based on an obsolete version of the code; it needs revision and may well be wrong.

State A fixed-size array of port objects. An index into this array is called a *port_position*.

Operations

- *MatchFind*(*device_pointer*) : returns either a *port_position* at which the port has has the state *Inactive* and in which the identifier matchs the identifier of the given device, or the information that there is no such position.
- *MatchMake*(*port_position*, *device_pointer*) : on entry, the port at the given *port_position* must have the state *Active*. The state of the port is then changed to *Owned*, and the device pointer is stored in the port.
- *MatchBreak*(*port_position*) : on entry, the port at the given *port_position* must have one of the states *Owned* and *Dead*. On exit, the state of the port is *Dead*.
- *OnControl*(*msg*) : The message specifies a port by *port_position*. Three and only three forms of message are possible (else an exception is raised).
  - *commandActive*
    Precondition. The port must have the state *Inactive*. There must be no device in the *DeviceList* with the same identifier, unless its state is *Inactive*. If there is another port in the *PortArray* with a similar[4] identifier, with a state different from *Active*, then the state of the port identified in the message is set to *Duplicated*. If on the other hand there is no such port, then the state of the port identified in the message is set to *Active*. If there is a device in the *DeviceList* with the same identifier, then the state of that device is set to *Active*, and the *port_position* in it is set to that specified in the message; the device pointer is stored in the port specified in the message; a *commandBuild* message is sent to the DSP. (There are some details to be filled in here – building and sending this message is just sketched in the code.) All of the above operations are performed while holding the lock (*matchLock*) that serialises access to the matching data structures[5].
  - *commandDead*
    Precondition. The port state must be other than *Inactive*[6]. If it is *Owned*,

[4]This means that the version field of the identifier may be different

[5]It is not clear to me that this lock need not be taken during operations such as (for example) *MatchFind*. If an exception is raised, then it is possible that the lock is released by some C++ voodoo, though the system state may not be consistent with its invariant.

[6]As far as I can see from the code, in fact (by an accident) the state must be either *Owned* or *Dead*

then the state of the device which is linked from it must be other than *Inactive*.

Should the port state be *Owned*, then "packets are deleted" – more detail is needed here. A *commandCleanup* message is sent to the DSP. Should the port state be *Duplicate*[7], then "the first duplicate is revived". The *SetState* procedure appears to lack a suitable code path to revive duplicates.

  – *commandRegisterDone*

    Precondition. The port state must be one of *Owned* or *Dead*. The message size must be 1.

    If the port's state is *Owned*, then "all in flight packets are time-stamped" – I have not investigated in detail what this means. The idea appears to be that one of the buffers associated with the device is put into a 'shutting-down' state. On the other hand, if the port's state is *Dead*, nothing is done.

- *OnRegPacket* : The message specifies a port by *port_position*. The state of the port must be *Owned* or *Dead*, and the message must have size 1.

  If the port's state is *Owned*, then "the state of the register packet is changed to finished, and a log message is written if no match is found in the map". (I have not investigated what this means.) On the other hand, if the port's state is *Dead*, nothing is done.

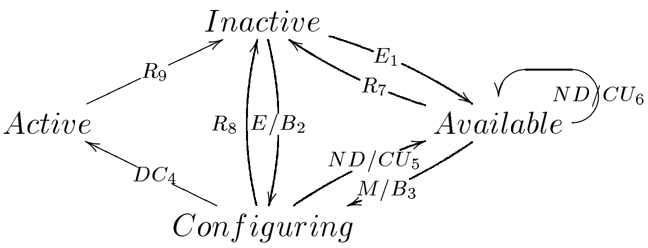
[7]Unless I am mistaken, this cannot arise.

9 PC Device

NEW(as of Mar 6 2002 10:25:43)

States Inactive
Available
Configuring
Active

State graph



Tabular form:

Transition number	*Start State*	*End State*	*stimulus*	*response*
1	Inactive	Available	Entry to device list, no matching available port	
2	Inactive	Configuring	Entry to device list, matching available port	Build
3	Available	Configuring	Matched port receives newID same id	Build
6	Available	Available	Matched port receives nowDead	CleanUp
7	Available	Inactive	Removal from device list	
4	Configuring	Active	Device configuration complete	
5	Configuring	Available	Matched port receives nowDead	CleanUp
8	Configuring	Inactive	Removal from device list	
9	Active	Inactive	Removal from device list	

Terminology

current, active A device is current or active if it is in the device list. (It then has a state-code different from Inactive.)

Invariants

1. If d and d' are different devices in the device list, then their identifiers differ in type-code or switch setting.

2. other invariants are in section 8 on page 17.

WARNING: I have omitted everything to do with '*RegPkt*'. It would be good to incorporate this. The important thing is that register reading and writing is done only when you have a device in an owned state, under the control of a handler.

WARNING: What appears below is based on the code which I have been looking at (Device.h, Device.cpp from Doug's 'Hakit' edition). It bears little relation to the version 2 description. Moreover, it seems to contain some detritus that is apparently not used – for example the state *Configured*. The code seems simplified with respect to the version 2 description.

It needs to be brought into line with the above table. (WARNING: It is wrong!)

States variables *portNum*
params
localID
(config) *level*

codes *Inactive* (Initial)
Active
Configured

Inputs

- $MatchDevice(port)$:
- $Process()$: Called on each lift of the message queue from the DSP
- $Kill()$: Called by owning handler on shutdown

Outputs

- $MesgBuild(params)$:
- $Reset()$:
- $Configure(level)$:

Transitions State machine - actions when not in specified states are fatal errors

- *Inactive*
  - On $MatchTest(ID)$ when $ID = localID$,
    $MesgBuild(params)$,
    set *level* to 0
    Next state: *Active*
- *Active*
  - On $Process()$
    Configure(level) ,
    Next state-code: *Configuring*
  - When $Configure()$ is true, ??????
    Next state-code: Configured

- On *UnMatch*() ,
  Reset() ,
  Next state-code: *Inactive*
- On *Destroy*() ,
  $port.UnMatch$,
  Reset() ,
  Next state-code: *Inactive*

• *Configured*

- On *UnMatch*() ,
  Reset() ,
  Inactive;
- On *Destroy*() ,
  $port.UnMatch$,
  Reset() ,
  Next state-code: *Inactive*

10 PC DeviceList

The device list is the set of current devices, both those that are and those that are not available for linking.

Terminology A physical ID is the ID reported by a physical device. It comprises a type code, a switch setting and a version number.

A device *tolerates* a range of physical ID's – those with a given type-code and switch setting, for which the firmware version number lies between certain limits.

Invariant The ID ranges tolerated by devices in the device list are disjoint. In other words, it cannot be the case that there are two (distinct) devices which both tolerate the same physical identifier. The devices must differ in at least one of type-code, and switch-setting.

WARNING: What appear below needs revision.

States

variables	*matchLock*	Lock – can get calls from HandlerIndex and PortArray
	ports	array of ports
	devices	array of (pointers to?) devices

codes ???? uninitialised/initialised ????

Inputs

- *Add*(*device*) Check that the ranges of ID's accepted by devices in the list would be disjoint.
- *Remove*(*device*) Find the device in the list. ??? Can we assume its there??? If the device state is *Owned*, break the Match (via port) associated with device, Force the device into the unmatched?? state.
- *MatchFind*(*ID*) If there's a device with the given ID on the list, which is matched, return true (and I presume, return the device found). If there's such a device, but not matched, throw an exception. If there's no device in the list with that ID, return false. (second arg appears to be unused: probably an output argument).

Outputs ?????? Code looks sketchy. Locking needs sorting out. Returns from procedures, etc etc.

Transitions ????

11 PC Handler

WARNING: To be revised.

A handler 'looks after' a number of devices[8]; presumably devices of the same type.

A handler (I THINK) has its own thread. (*Process*.)

The following concerns only handlers-in-general. The *Handler* class is a base class from which specific handlers are derived: for example an OPD 'joint' handler, or a modem handler.

States Each handler has an array of pointers to devices.

variables	*index*	what is this ????
	devices	array of devices. Only state variable mentioned in generic code
	build	something to do with Share????

codes Actually, there don't seem to be any codes though they are listed in the rough document. What does this mean??? Guess made in transitions below.

Dead	unbuilt?? (Initial)
Alive	built ??

Inputs Commands to a handler

- *Dead* – ??? On successful *Construct* moves to *Alive*
- *Alive*
  - *Build*(*devno*) – ?? stubbed out. Initialisation?
  - *Kill*() – ?? stubbed out. ??? On successful Kill, moves to *Dead*????
  - *Get* – returns number of devices under control of this handler. Read only.
  - *GetDevice*(*devno*) – returns pointer to *devno*' th device. Read only.
  - *OnData*(*mesg*) – ???? Should be virtual
  - *OnTransfer*(*mesg*) – ???? Should be virtual.

Outputs ??

Transitions

- *Dead* – does this mean un-built??
  - *Build*(*devno*) : procedure
    ????
- *Alive*
  - *Kill*() : procedure
    ????

[8]'Device' isn't quite the right word; in the rest of this document, 'device' has the connotation of DSP device. If I understand Doug's plans correctly, one might have a PC handler for a radio modem.

- *Get* – returns number of devices under control of this handler.
- *GetDevice*(*devno*) – returns pointer to *devno*' th device.
- *OnData*(*mesg*) – ???? Should be virtual
- *OnTransfer*(*mesg*) – ???? Should be virtual.

12 PC Handler index

WARNING: To be revised.

The handler index is a dynamic array of indices, which are slots for (pointers to) handlers. The slots start off as *Empty*, become *Active* (through *Add*), and finish up as *Dead* (through *Remove*). If no slot is free when one is required, then a new slot is allocated and added to the end of the array. Either a handler is accessible through a slot or it isn't. Whenever a message is sent to a handler, the handler is identified by its position in the array.

> *How to describe Process – resumes the thread associated with each handler in the array*

State

variables *indices* Dynamic array of indices
devices device list

codes ????
Dead unbuilt?? (Initial)
Alive built ??

Inputs

- *Construct*(*device_list*) – initialise
- *Add*(*handler_pointer* – add a new handler, returns slot number
- *Remove*(n) – remove n'th handler
- *Message* : A control message is used to release[9] a handler index.
  There are also *Data* and *Transfer* messages.

Outputs ??? later ...

Transitions

- *Alive*
  - *Add*(*handler_pointer* – add a new handler, returns slot number
  - *Remove*(n) – remove n'th handler. Removes all devices; kills the handler ; disables index. ??? Some sanity checking necessary. (Can it be called when already removed?)
  - On Control – used only to release handler index. Stubbed out ???
  - On Data – check slot not empty (exception); if Dead, do nothing ; else pass through to handler.

[9] What does release mean?

  - On Transfer – ???
- *Dead*
  - *Construct*(*device_list*) – initialise *devices*, move to *Alive*.