cardres

COLLABORATORS

	TITLE : cardres	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 19, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

cardres

1.1 cardres.doc

```
    BeginCardAccess()
    CardAccessSpeed()
    CardChangeCount()
    CardForceChange()
    CardInterface()
    CardMiscControl()
    CardProgramVoltage()
    CardResetCard()
    CardResetRemove()
    CopyTuple()
    DeviceTuple()
    EndCardAccess()
    GetCardMap()
    IfAmigaXIP()
    OwnCard()
    ReadCardStatus()
    ReleaseCard()
```

1.2 card.resource/BeginCardAccess

```
 NAME
BeginCardAccess -- Called before you begin credit-card memory access

 SYNOPSIS
result=BeginCardAccess( handle )
d0      a1

BOOL BeginCardAccess( struct CardHandle * );

 FUNCTION
This function should be called before you begin access
to credit-card memory.

Its effect will depend on the type of Amiga machine your
code happens to be running on.  On some machines it
```

```
will cause an access light to be turned ON.

 INPUTS
handle - Same handle as that used when OwnCard() was called.

 RETURNS
TRUE if you are still the owner of the credit-card, and
memory access is permitted.  FALSE if you are no longer
the owner of the credit-card (usually indicating that
the card was removed).

 NOTES
This function may be called from within a task, or from a level 1
or level 2 interrupt.

It is highly recommended that you call this function
before accessing credit-card memory, as well as checking
the return value.  If it is a return value of FALSE, you
should stop accessing credit-card memory.

 SEE ALSO
OwnCard(), EndCardAccess()
```

1.3 card.resource/CardAccessSpeed

```
 NAME
CardAccessSpeed -- Select best possible memory access speed.

 SYNOPSIS
result=CardAccessSpeed( handle, nanoseconds );
d0       a1      d0

ULONG CardAccessSpeed( struct CardHandle *, ULONG );

 FUNCTION
This function is used to set memory access speed for all CPU
accesses to card memory.

Typically this information would be determined by first examining
the Card Information Structure.

Then you would use this function to let the card.resource
select the best possible access speed for you, however note
that the range of possible access speeds may vary on some
machines (depending on the type of credit-card interface
hardware being provided).

 INPUTS
handle - Same handle as that used when OwnCard() was called.

nanoseconds - Preferred access speed in nanoseconds.

 RETURNS
Speed - Access speed selected by resource (in nanoseconds).
```

```
0  - Not successful.  Either because the credit-card was
removed, or the access speed you requested is slower than
that supported by the credit-card interface hardware.

 NOTES
This function may be called from within a task, or from a level 1
or level 2 interrupt.

 SEE ALSO
OwnCard()
```

1.4 card.resource/CardChangeCount

```
 NAME
CardChangeCount -- Obtain card change count.

 SYNOPSIS
count = CardChangeCount( VOID )
d0

ULONG CardChangeChange( VOID );

 FUNCTION
This function returns the card change count.  The
counter is incremented by one for every removal, and
for every successful insertion (a card which is inserted
long enough to be debounced before it is removed again).

 RESULT
The change count number.

 NOTES
This function may be called from a task, or any level interrupt.

 SEE ALSO
```

1.5 card.resource/CardForceChange

```
 NAME
CardForceChange -- Force a card change.

 SYNOPSIS
success = CardForceChange( VOID )
d0

BOOL CardForceChange( VOID );

 FUNCTION
This function is not intended for general use.  Its
purpose is to force a credit-card change as if
the user had removed, or inserted a card.
```

```
This function is intended to be used by a utility program
which needs to force the current card owner to release
ownership of the card, thereby allowing the utility an
opportunity to own the credit-card.

 RESULT
TRUE if the function succeeded, FALSE if card change is
not allowed.  This function will generally succeed, unless
someone is using the card in reset remove mode at the time
this function is called.

 NOTES
This function should only be called from a task.

 SEE ALSO
```

1.6 card.resource/CardInterface

```
 NAME
CardInterface -- Determine the type of card interface.

 SYNOPSIS
return = CardInterface()
d0

ULONG CardInterface( void );

 FUNCTION
This function is used to determine the type of credit-card
(hardware) interface available.  For the most part the
card.resource hides the hardware details from devices within its
function calls.  However should we need to provide a work-around
because of differences, or limitations imposed by future interface
hardware, this function must be used to identify which interface
is available.

 RETURN
A ULONG value as defined in card.h/i.

 NOTES
In general only I/O devices (e.g., a device which interfaces
with a modem card) would need to provide work-arounds,
or alternative code.  An example would be a change in the way
interrupt requests from the card are handled.  Specific details
will be provided as need in the future.  I/O devices) should abort
properly if this function returns a value which is unknown.

Current implementations (see card.h/i) -

CARD_INTERFACE_AMIGA_0
------------------------------------------------------------

The card slot can be configured for use as an I/O interface
by using the CardMiscControl() function.
```

```
The card slot inhibits writes to cards which do not negate
the WP status bit.  This can be overridden by using the
CardMiscControl() function.

Changes in the interrupt request line are latched by a gate-array,
and have to be obtained via the status change mechanism provided
when you call the OwnCard() function.  The interrupt is cleared
when you return from the status change interrupt.  A level 2
interrupt is generated.  Usually you will want to clear the
interrupt on the card at this time, and Signal() a task.  The IRQ
line is the same as the RDY/BSY line.

Changes in BVD1, WP, and RDY/BSY are also latched by the gate-array,
and are obtainable via the status change mechanism provided by
the OwnCard() function.  A level 2 interrupt is generated.

Changes in BVD2 (also used for digital audio) have to be
monitored via polling.  Generally this will cause no problem.
Monitoring changes in BVD1 & BVD2 to monitor for low battery
condition can be handled by a low priority tool which periodically
checks the condition of both lines using the ReadCardStatus()
function.

As of card.resource V39 (check VERSION in resource base), the
CardMiscControl() function can be used to enable/disable
status change interrupts for changes in BVD1, BVD2, and the
RDY/BSY status line.  Status change interrupts for WR
(Write-protect enable/disable) are always enabled.  The default
state of enabled/disabled status change interrupts noted above
are unchanged, and automatically reset to the defaults when
a card is removed, or when even a task releases ownership
of the card.

Some PC oriented eight (8) bit cards may require you read
odd-byte I/O address registers at the corresponding even-byte
address plus 64K.  There is sufficient I/O address space
provided that exceeding I/O address space should not be a problem.

Your code should wait at least 1 millisecond for Vpp to stabilize
after voltage change (see CardProgramVoltage()).

  SEE ALSO
CardMiscControl(), resources/card.i, resources/card.h
```

1.7 card.resource/CardMiscControl

```
 NAME
CardMiscControl -- Set/Clear miscellaneous control bits

 SYNOPSIS
control_bits=CardMiscControl( handle, control_bits );
d0        a1        d1

UBYTE CardMiscControl( struct CardHandle *, UBYTE );
```

```
 FUNCTION
Used to set/clear miscellaneous control bits (generally for
use with I/O cards).

 INPUTS
handle - Same handle as that used when OwnCard() was called.

control_bits - A mask value of control bits to be turned on/off.

The bit values which might be usable are defined in card.h/i.

For example, to enable digital audio, and disable hardware
write-protect (if supported), you would call this function with these
values --

CARDF_DISABLE_WP|CARDF_ENABLE_DIGAUDIO

Then to turn off digital audio, but leave write-protect
disable, you would use a value of --

CARDF_DISABLE_WP

Finally too reenable write protect, call this function with
a mask value of 0.

 RETURNS
control_bits - The same mask value you called this function
with if successful.  If one, or more bits has been cleared
in the return mask, this would indicate that the control bit
is not being supported, or that the card has been removed
by the user.

For example, if you called this function with a mask value
of --

CARDF_DISABLE_WP|CARDF_ENABLE_DIGAUDIO

And this function returned a value of --

CARDF_DISABLE_WP

This would indicate that it is not possible to enable digital
audio (most likely because this feature has not been implemented).

 NOTES
This function may be called from within a task, or from a level 1
or level 2 interrupt.

!!!IMPORTANT!!!

You should ALWAYS try to enable digital audio for I/O cards
as this will also configure the card socket for the I/O
interface (if supported).

Not all cards will connect the write-enable line (e.g.,
some I/O cards).  On some machines (e.g., the A600) it will
not be possible to write to such cards unless you disable
```

```
write-protection by using this function.

!!!NEW!!!

For card.resource V39 (check resource base for VERSION before
using), new bits have been defined which let you enable/disable
particular status change interrupts.  See CardInterface() for
defaults.  These new bits are backwards compatable with V37
for which only the CARDB_DISABLE_WP, and CARDB_ENABLE_DIGAUDIO
bits were defined.  These new bits allow you to enable, or
disable specific status change interrupts including BVD1/SC,
BVD2/DA, and BSY/IRQ.  The defaults for these status change
interrupts are unchanged from V37, and WR (Write-protect) status
change interrupts are always enabled as they use to be.

An example of use:

CARD_INTF_SETCLR!CARD_INTF_BVD1

Would enable BVD1/SC status change interrupts, and not change
the enable/disable state for BVD2/DA or BSY/IRQ status change
interrupts.  If the change was made successfully, the
CARD_INTB_BVD1 bit would be set in register D0 when this function
returns.

 SEE ALSO
CardInterface(), resources/card.h, resources/card.i
```

1.8 card.resource/CardProgramVoltage

```
 NAME
CardProgramVoltage -- Set programming voltage.

 SYNOPSIS
success=CardProgramVoltage( handle, voltage );
        a1      d0

LONG CardProgramVoltage( struct CardHandle *, ULONG );

 FUNCTION
Used to set programming voltages (e.g., for FLASH-ROM/EPROM
cards).

 INPUTS
handle - Same handle as that used when OwnCard() was called.

voltage - See card.i/h for valid values.

 RETURNS
1  - Successful.

0  - Not successful.  Most likely because the credit-card
card has been removed, and you are no longer the owner.

-1 - This function is not being supported.  On some machines
```

```
with a minimal (hardware) credit-card interface, this feature
may not be possible.

 NOTES
This function may be called from within a task, or from a level 1
or level 2 interrupt.

     !!!WARNING!!!

Flash-ROM programming requires careful coding to prevent
leaving the Erase command on too long.  Failure to observe
the maximum time between the Erase command, and the Erase-Verify
command can make a Flash-ROM card unusable.  Some Flash-ROM cards
may provide an internal watch-dog timer which protects the card.

Because of the relatively long time (e.g., 10ms) between Erase, and
Erase-Verify which must be observed, the need for such critical
timing can be problematic on a multi-tasking machine.

Vendors of Flash-ROM's recommend a high priority interrupt
generated by a 10ms timer be used to turn off Erase.  On the
Amiga this can be accomplished by using a CIA-B interval timer.
The timer.device also provides a mechanism for generating a low
priority interrupt.  The timer.device is easier to use than CIA
interval timers, though not as accurate or as safe.

Even if the Flash-ROM card provides an internal watch-dog timer,
implementation of the code during Erase should assume that
the Flash-ROM does not.

 SEE ALSO
OwnCard(), resources/card.h, resources/card.i
```

1.9 card.resource/CardResetCard

```
 NAME
CardResetCard -- Reset credit-card.

 SYNOPSIS
success=CardResetCard( handle );
      a1

BOOL CardResetCard( struct CardHandle * );

 FUNCTION
Used to reset a credit-card.  Some cards, such as some
configurable cards can be reset.

Asserts credit-card reset for at least 10us.

 INPUTS
handle - Same handle as that used when OwnCard() was called.

 RETURNS
TRUE  - Successful.
```

```
FALSE  - Not successful.  Most likely because the credit-card
card has been removed, and you are no longer the owner.

 NOTES
This function may be called from within a task, or from a level 1
or level 2 interrupt.

***IMPORTANT***

It is the responsibility of the card owner to reset
configurable cards, or any other type of card such as
some I/O cards before calling ReleaseCard() if the owner
has made use of that card such that it is no longer in its
reset state (unless you are releasing the card because it
has been removed).

If the card manufacturer indicates that a certain amount
of time must elapse between end of reset, and completion
of card initialization, you should wait at least that long
before releasing the card (unless you are releasing the card
because it has been removed).

 SEE ALSO
OwnCard(), ReleaseCard()
```

1.10 card.resource/CardResetRemove

```
 NAME
CardResetRemove -- Set/Clear reset on card removal.

 SYNOPSIS
success=CardResetRemove( handle, flag );
       a1      d0

BOOL CardResetRemove( struct CardHandle *, ULONG );

 FUNCTION
Used to set/clear HARDWARE RESET on card change detect.

This function should generally not be used by devices
which support HOT-REMOVAL.  HARDWARE RESET on removal
is generally intended for execute-in-place software, or
ram cards whose memory has been added as system ram.

 INPUTS
handle - Same handle as that used when OwnCard() was called.

flag - TRUE if you want to SET HARDWARE RESET on credit
       card removal.  FALSE if you want to CLEAR HARDWARE
       RESET.

 RETURNS
1  - Success.
```

```
0  - Function failed (most likely because the card was removed
     by the user, and you are no longer the owner of the card).

-1 - This function is not being made available.

 NOTES
This function should only be called from a task.

 SEE ALSO
OwnCard()
```

1.11 card.resource/CopyTuple

```
 NAME
CopyTuple -- Find/copy a credit card tuple.

 SYNOPSIS
success = CopyTuple( CardHandle, buffer, tuplecode, size );
d0      a1   a0  d1     d0

BOOL CopyTuple( struct CardHandle *, UBYTE *, ULONG, ULONG );

 FUNCTION
This function scans credit-card memory for a tuple, and
if found, copies it into a supplied buffer.  The entire
tuple (including the tuple code, and link) are copied to
the supplied buffer.  The number of bytes copied to the
buffer will be 2 bytes, plus the argument "size", or the
value in the tuple LINK field (whichever is SMALLER).

The software calling this function is responsible for
examining the copy of the tuple (e.g., recognition of
fields, recognition of stop bytes, etc. within the tuple).

This function does the best job it can to find a tuple
in attribute, or common memory.  It follows tuple chains,
and skips odd bytes in attribute memory.

This function monitors for credit-card removal while reading data.
If the credit-card is removed while a byte is being read, it will
stop searching, and return FALSE.

This function does not protect against another task writing
to credit-card memory while data is being read.  The device
is responsible for single-threading reads/writes to the
credit card as needed.

This function can be used to find multiple tuple codes; this
is a very rare case, so the mechanism provided for doing so is
unusual.  See INPUTS below for more information.

This function does not read bytes within the body of any tuples
except for the tuple you want copied, and the basic compatibility
tuples this function understands (see list below).
```

On some machines this function may slow down memory access speed while reading the tuple chain. This is done to prevent potential problems on slow cards. By examining the CISTPL_DEVICE, and CISTPL_DEVICE_A tuples, you can determine the best possible memory access speed for the type of card inserted. Because memory access speed may be slowed down, calls to this function should be single-threaded.

The Card Information Structure must start with a CISTPL_DEVICE tuple stored as the first tuple in attribute memory. If not, this function will search for a CISTPL_LINKTARGET tuple stored at byte 0 of common memory. Therefore it is possible to store a CIS on cards which do not have any writeable attribute memory, though this may cause problems for other software implemented on other machines. For example, some SRAM cards do not come with writeable attribute memory, and/or some may have OPTIONAL EEPROM memory which may not have been initialized by the card manufacturer. While it could be argued that such cards do not conform to the PCMCIA PC Card Standard, such cards are cheaper, and therefore likely to be used.

INPUTS
CardHandle - Same handle as that used when OwnCard() was called.

buffer - Pointer to a buffer where the tuple will be copied.

The buffer should be at least as large as "size" + 8 (see use of "size" argument below). Therefore the minimum buffer size is 8 bytes. See NOTES below.

tuplecode - The tuple code you want to search for. This is a ULONG value. The upper 16 bits should be 0, or a number between 1-32K where a value of 0 means you want to find the first tuple match, a value of 1 the second, etc. For example -

0x41 means find the FIRST tuple equal to $41.

((1<<16)|(0x41)) means find the SECOND tuple equal to $41.

((2<<16)|(0x41)) means find the THIRD tuple equal to $41.

size - The maximum number of bytes you want copied (not including the tuple code, and link). The actual number of bytes copied may be less than "size" if the tuple link field is less than "size".

A size of 0 will result in only the tuple code, and link being copied to your buffer if the tuple is found.

If you do not care how many bytes are copied, any unsigned value of 255 or greater will do. In this case a maximum of 257 bytes might be copied to your buffer (if the tuple link field is the maximum of 255).

Other sizes are useful if you know the size of the tuple

```
       you want copied, or you know there are active registers
       stored within the tuple, and only want to copy a portion
       of a tuple.

 RETURNS

TRUE if the tuple was found, and copied, else FALSE.
This function may also return false if the CIS is believed
to be corrupt, or if the card is removed while reading the
tuples.

 NOTES

This function can be called multiple times (adjusting the "size"
argument) to read a tuple of variable length, or unknown size.

Your supplied buffered is used by this function for working
storage - the contents of it will be modified even if this
function fails to find the tuple you want a copy of.

This function should NOT be used to find/copy tuples of type :

- CISTPL_LONGLINK_A
- CISTPL_LONGLINK_C
- CISTPL_NO_LINK
- CISTPL_LINKTARGET
- CISTPL_NULL
- CISTPL_END

These tuples are automatically handled for you by this function.

 SEE ALSO
OwnCard()
```

1.12 card.resource/DeviceTuple

```
 NAME
DeviceTuple -- Decode a device tuple

 SYNOPSIS
return=DeviceTuple( tuple_data, storage)
          a0    a1

ULONG DeviceTuple( UBYTE *, struct DeviceTData *);

 FUNCTION
Extracts SIZE, TYPE, and SPEED from a device tuple (generally
obtained with CopyTuple()).

 INPUTS
tuple_data - Pointer to a CISTPL_DEVICE tuple (generally obtained
with CopyTuple()).

storage - Pointer to a DeviceTData structure in which results
are to be stored.
```

```
struct  DeviceTData {
  ULONG dtd_DTsize;   /* Size of card (bytes) */
  ULONG dtd_DTspeed;    /* Speed in nanoseconds */
  UBYTE dtd_DTtype;   /* Type of card   */
  UBYTE dtd_DTflags;    /* Other flags    */
};
```

```
 RETURN
SIZE (same as dtd_DTsize) if the Device Tuple could be decoded.
FALSE (0) if the tuple is believed to be invalid.  The tuple is
considered to be invalid if:

  The tuple link value is 0.

  The device type/speed byte is $00, or $FF.

  The device type is DTYPE_EXTEND, which is undefined
  as of this writing.

  The extended speed byte is a value which is
  undefined as of this writing.

  The extended speed byte is extended again which is
  undefined as of this writing.

  The device Size byte is $FF.

 NOTES
Some cards may not have a size specified in the device
tuple.  An example would be an I/O card.  The size would be
returned as one (1) in this case.

You should not call this function with a partial CISTPL_DEVICE
tuple, or the return values may be junk.

 SEE ALSO
CopyTuple(), resources/card.h, resources/card.i
```

1.13 card.resource/EndCardAccess

```
 NAME
EndCardAccess -- Called at the end of credit-card memory access

 SYNOPSIS
result=EndCardAccess( handle )
d0      a1

ULONG EndCardAccess( struct CardHandle * );

 FUNCTION
This function should be called when you are done accessing
credit-card memory.
```

```
Its effect will depend on the type of Amiga machine your
code happens to be running on.  On some machines it
will cause an access light to be turned OFF in approximately
1/2 second.

On machines which support an access light, the light will
automatically be turned off when you call ReleaseCard().

 INPUTS
handle - Same handle as that used when OwnCard() was called.

 RETURNS
TRUE if you are still the owner of the credit-card.  FALSE
if you are no longer the owner of the credit-card (usually
indicating the card was removed).

 NOTES
This function may be called from within a task, or from a level 1
or level 2 interrupt.

It is highly recommended that you call this function
after accessing credit-card memory, as well as checking
the return value.  If it is a return value of FALSE, you
should stop accessing credit-card memory, and conclude
that the card was removed before this function was called.

On some machines it is possible that the credit-card will be
removed before you receive the removed interrupt.

 SEE ALSO
OwnCard(), ReleaseCard(), BeginCardAccess()
```

1.14 card.resource/GetCardMap

```
 NAME
GetCardMap -- Obtain pointer to CardMemoryMap structure

 SYNOPSIS
pointer=GetCardMap()
d0

struct CardMemoryMap *GetCardMap( void );

 FUNCTION
Obtain pointer to a CardMemoryMap structure.  The structure
is READ only.

Devices should never assume credit-card memory appears
at any particular place in memory.  By using this function
to obtain pointers to the base memory locations of the various
credit-card memory types, your device will continue to work
properly should credit cards appear in different memory
locations in future hardware.

 RETURNS
```

```
Pointer to CardMemoryMap structure -

  struct CardMemoryMap {
  UBYTE *cmm_CommonMemory;
  UBYTE *cmm_AttributeMemory;
  UBYTE *cmm_IOMemory;
  };

As of card.resource V39, this structure has been extended to
include the size of these memory regions.  See card.h/card.i
for the new fields.  If card.resource V39, use the constants
in the CardMemoryMap structure rather than hard coded constants
for memory region size.

 NOTES
If any pointer in the structure is NULL, it means this type
of credit-card memory is not being made available.

 SEE ALSO
resources/card.h, resources/card.i
```

1.15 card.resource/IfAmigaXIP

```
 NAME
IfAmigaXIP -- Check if a card is an Amiga execute-in-place card.

 SYNOPSIS
result=IfAmigaXIP( handle )
d0           a2

struct Resident *IfAmigaXIP( struct CardHandle * );

 FUNCTION
Check to see if a card in the slot is an Amiga execute-in-place
card.  The Card Information Structure must have a valid
CISTPL_AMIGAXIP tuple.

Tuples can be treated like structures.  The format of a
CISTPL_AMIGAXIP tuple is -


struct TP_AmigaXIP {
  UBYTE TPL_CODE;
  UBYTE TPL_LINK;
  UBYTE TP_XIPLOC[4];
  UBYTE TP_XIPFLAGS;
  UBYTE TP_XIPRESRV;
};


The TPL_CODE field must be equal to CISTPL_AMIGAXIP (0x91).

The TPL_LINK field must be equal to the length of the body
of a CISTPL_AMIGAXIP tuple (0x06).
```

The TP_XIPLOC array is the memory location of your ROM-TAG stored in little-endian order. This value is stored as an "offset" into common memory as is the standard for storing 32 bit bit pointers in tuples.

For example, a pointer to a ROM-TAG stored at an offset of 0x00000200 would be stored as four bytes like so:

```
0x00, 0x02, 0x00, 0x00
```

Currently credit-card common memory is mapped starting at memory location 0x600000. Because a ROM-TAG is used, it is implied that execute-in-place code can be compiled/linked to use absolute references. It is believed that most developers will not want to have to write pc-relative code only.

The TP_XIPFLAGS field is treated as a set of flag bits. See card.i/h for defined bits. All undefined bits MUST be 0.

The TP_XIPRESRV field must be 0 for now.

The system polls for cards which have a CISTPL_AMIGAXIP tuple at the same time that it searches for devices to boot off. When a card with a valid CISTPL_AMIGAXIP tuple is found, the system will call your ROM-TAG via Exec's InitResident() function.

The system examines the return code from InitResident(). A NULL return in D0 means you are done with the card, and it can be removed by the user. A successful return indicates you are still using the card. Some programs (e.g., some games) may never return. The only requirement is that if you do return, you must leave the system in a "good" state (including returning most of, or all the memory you allocated). The standard convention for preserving registers apply.

Note that your execute-in-place code will not be called a second time, unless you have returned a non-successful result. In this case your execute-in-place code MUST assume the user can remove, and insert your card again. There are a variety of ways to check for re-insertion (e.g., search for a message port, device, library, task, etc., that you created).

Note that your execute-in-place code runs in an environment similar to boot block games; before DOS has been initialized!

Your execute-in-place code should NOT try to initialize DOS because DOS requires a suitable disk-like device be at the head of the expansion base mountlist to boot off.

If you need DOS, it is possible to boot off a credit-card using carddisk.device. Such cards will require a valid CISTPL_DEVICE tuple, and CISTPL_FORMAT tuple. A portion of the card can be used for a minimal number of data blocks like the method described above.

```
However this method is not recommended, though it is anticipated
that some developers will have thought of, and used this
method anyway.  If you must do this, at least use the
CardHandle returned by OwnCard() to set hardware reset
on removal else the machine will likely crash anyway if
the card is removed while your execute-in-place code is running.

 RETURNS
Pointer to a ROM-TAG on the card, or NULL indicating that:

o The card does not meet the above requirements, or
o The card has been removed, or
o You are not the owner of the credit-card.

 NOTES
This function is being made public so developers can test
test that their execute-in-place credit-cards meet the
requirements for an Amiga execute-in-place card.  In general
there is no reason for devices, or applications to use this
function.

Amiga execute-in-place software is identified via a tuple code
reserved for manufacturer specific use, therefore the manufacturer
of the card may freely use the CISTPL_VERS_1, or CISTPL_VERS_2
tuples to place identification information on the credit-card.

No attempt has been made to make use of the MS-DOS execute-in-place
method; it is believed that most manufacturers of Amiga
execute-in-place software will prefer a simple, and small
scheme for running their execute-in-place code.

 SEE ALSO
OwnCard(), resources/card.h, resources/card.i
```

1.16 card.resource/OwnCard

```
 NAME
OwnCard -- Own credit card registers, and memory

 SYNOPSIS
return = OwnCard( handle )
d0    a1

struct CardHandle *OwnCard( struct CardHandle * );

 FUNCTION
This function is used to obtain immediate, or deferred
ownership of a credit-card in the credit-card slot.

Typically an EXEC STYLE DEVICE will be written to interface
between an application, and a credit card in the slot.  While
applications, and libraries can attempt to own a credit-card
in the card slot, the rest of this documentation assumes a
device interface will be used.
```

```
Because credit-cards can be inserted, or removed by the user at
any time (otherwise known as HOT-INSERTION, and HOT-REMOVAL),
the card.resource provides devices with a protocol which
lets many devices bid for ownership of a newly inserted card.

In general, devices should support HOT-REMOVAL, however there
are legitimate cases where HOT-REMOVAL is not practical.  For
these cases this function allows you to own the resource using
the CARDB_RESETREMOVE flag.  If the card is removed before your
device calls ReleaseCard(), the machine will RESET.

 INPUTS
handle - pointer to a CardHandle structure.

  struct CardHandle {
  struct Node      cah_CardNode;
  struct Interrupt  *cah_CardRemoved;
  struct Interrupt  *cah_CardInserted;
  struct Interrupt  *cah_CardStatus;
  UBYTE      cah_CardFlags;
  };

The following fields in the structure must be filled
in by the application before calling OwnCard() -

cah_CardNode.ln_Pri -

  See table below.  The Node field is used by the resource to
  add your handle to a sorted list of CardHandle structures.
  This list is used by the resource to notify devices when the
  device owns the credit-card.

  Your device will only be notified (at most) one time
  per card insertion, and perhaps less often if some
  higher priority device on the notification list retains
  ownership of a card in the slot.

Priority  Comments
-------------------------------------------------
  >= 21 Reserved for future use

  10-20 To be used by third party devices (e.g.,
    I/O CARD manufacturers) which look for
    specific card tuples to identify credit-cards.

  01-19 Reserved for future use

  00  To be used by general purpose devices which
    have loose card specification requirements.

  <= -1 Reserved for future use


cah_CardNode.ln_Type -

  Must be set to 0 for now.  This field may be used in the
```

```
    future to identify an extended CardHandle structure.

  cah_CardNode.ln_Name -

    Must be initialized to NULL, or name of device which owns
    this structure.

  cah_CardRemoved -

    Pointer to an initialized interrupt structure.  Only the
    is_Data, and is_Code fields need to be initialized.  This
    is the interrupt code which will be called when a credit-card
    which your device owns is removed.  Once you receive this
    interrupt, all credit-card interface control registers are
    reset (e.g., programming voltage, access speed, etc.), and
    you should stop accessing the card as soon as possible.

    Because your code is called on interrupt time, you
    should do the least amount possible, and use little
    stack space.

    This pointer can be NULL if you have asked for reset
    on card-removal, and you never turn reset off.

  cah_ CardInserted -

    Pointer to an initialized interrupt structure.  Only the
    is_Data, and is_Code fields need to be initialized.  This
    is the code which will be called when your CardHandle owns
    the credit-card in the slot.

    Note that your code may be called on the context of
    an interrupt, or a task in FORBID, therefore you should
    do the least amount possible, and use little stack space.

    Note that it is possible to receive a card removed
    interrupt immediately before you receive this interrupt if
    the card is removed while your CardInserted interrupt
    is being called.

    Your device owns the credit-card until the card is manually
    removed by the user, or you release the card by calling
    ReleaseCard().

    Your device should examine the card in the slot (e.g., look
    for specific tuples), and decide if the card is of a type your
    device understands.

    If not, release ownership of the card by calling
    ReleaseCard() so that other devices will be given a chance to
    examine the current card in the credit-card slot.

  cah_CardStatus -

    Pointer to an initialized interrupt structure.  Only the
    is_Data, and is_Code fields need to be initialized.
```

```
    Note that your code will be called on the context of
    an interrupt, therefore you should do the least
    amount possible, and use little stack space.

    Note that it is possible to receive a card removed
    interrupt immediately before you receive this interrupt
    if the card is removed during this interrupt.

    If this pointer is NULL, you will not receive card status
    change interrupts.

    Your interrupt code will be called with a mask value in
    register D0, and a pointer to your data in A1.

    The mask value in D0 can be interpreted using the same bit
    definitions returned by ReadCardStatus().  Note that more
    than one bit may be set, and the mask only tells you what has
    changed, not the current state.

    Use ReadCardStatus() if you need to determine the current
    state of the status bits.

    Not all status change interrupts will necessarily be
    enabled on all systems.  For example, on some systems
    BVD2/DA status change interrupts will not be enabled so
    that digital audio can occur without generating many
    interrupts.  Status change interrupts are typically meant
    to be used for monitoring BSY/IRQ, WR, and BVD1/SC.  Battery
    voltage low detection would best be done by a separate
    utility which periodically polls BVD1 & BVD2 by using the
    ReadCardStatus() function.

    Typically the mask value in D0 MUST be returned unchanged
    on exit from your code.  The return value in D0 is then used
    to clear the source(s) of the interrupt.

    In the rare case that you need to keep a status change
    interrupt active, clear the appropriate bit(s) in D0 before
    returning via RTS.  Clear no bits other than those defined
    as valid bits for ReadCardStatus()!

    !!!NEW FOR V39!!!

    See definition of CARDB_POSTSTATUS below.

  cah_CardFlags -

    Optional flags (all other bits must be 0).

    - CARDB_RESETREMOVE means you want the machine to
      reset if the credit-card in the slot is removed
      while you own the credit-card.

    - CARDB_IFAVAILABLE means you only want ownership of
      the credit-card in the slot if it is immediately
      available.
```

If it is available, your CardHandle structure will be added to a list so that you can be notified via an interrupt when the credit-card is removed by the user.

If the credit-card is not immediately available (either because there is no credit-card in the slot, or because some other device owns the credit-card), your CardHandle structure will NOT be added to the notification list.

- CARDB_DELAYOWNERSHIP means you never want a successful return from OwnCard() even if the credit-card is available. Rather you will be notified of ownership via your cah_CardInserted interrupt. If you use this flag, OwnCard() will always return -1. This flag cannot be used with the CARDB_IFAVAILABLE flag.

- CARDB_POSTSTATUS is new for V39 card.resource (check resource base VERSION before using). It is meant to be used by drivers which want to service the card hardware AFTER the status change interrupt has been cleared on the gate array. Previously a PORTS interrupt server had to be added to do this; this is somewhat more efficient, and easier to use. Your status change interrupt is first called with status change bits in register D0. You would examine these bits, and set a flag(s) for the POST callback. When you return from the status change interrupt, the interrupt on the gate array is cleared (based on what you return in register D0), and your status change interrupt is immediately called again, but this time with 0 in D0. The value you return in D0 for the POST callback case is ignored.

ALL other fields are used by the resource, and no fields in the structure may be modified while the structure is in use by the resource. If you need to make changes, you must remove your CardHandle (see ReleaseCard()), make the changes, and then call OwnCard() again.

RESULTS

0 - indicates success, your device owns the credit card.

-1 - indicates that the card cannot be owned (most likely because there is no card in the credit card slot).

ptr - indicates failure. Returns pointer to the CardHandle structure which owns the credit card.

NOTES

This function should only be called from a task.

CardHandle interrupts are called with a pointer to your data in A1, and a pointer to your code in A5. With the exception of status change interrupts, D0-D1, A0-A1, and A5-A6 may be treated as scratch registers. Status change interrupts are also called with meaningful data in D0, and expect D0 be preserved upon RTS from your code. No other registers are guaranteed to contain initialized data. All other registers

```
must be preserved.

  SEE ALSO
ReleaseCard(), ReadCardStatus(), resources/card.i, resources/card.h
```

1.17 card.resource/ReadCardStatus

```
 NAME
ReadCardStatus -- Read credit card status register

 SYNOPSIS
status=ReadCardStatus()
d0

UBYTE ReadCardStatus( void );

 FUNCTION
Returns current state of the credit card status register.

See card.h/i for bit definitions.

Note that the meaning of the returned status bits may vary
depending on the type of card inserted in the slot, and
mode of operation.  Interpretation of the bits is left
up to the application.

 RETURNS
A UBYTE value to be interpreted as status bits.

 NOTES
This function may be called from within a task, or from any level
interrupt.

 SEE ALSO
resources/card.h, resources/card.i
```

1.18 card.resource/ReleaseCard

```
 NAME
ReleaseCard -- Release ownership of credit card

 SYNOPSIS
ReleaseCard( handle, flags )
       a1      d0

void ReleaseCard( struct CardHandle *, ULONG );

 FUNCTION
This function releases ownership of the credit card in the
slot.

The access light (if any) is automatically turned off
```

```
(if it was turned on) when you release ownership of
a card you owned, and all credit-card control registers
are reset to their default state.

You must call this function if -

You own the credit-card, and want to release it so that
other devices on the notification list will have a chance
to examine the credit-card in the card slot.

You took a Card Removed interrupt while you owned the
credit-card.  If so, you MUST call this function, else
no other task will be notified of newly inserted cards.  On
some machines the credit-card interface hardware may also
be left disabled until you respond to the card removed interrupt
by calling this function.

You want to remove yourself from the notification list (see
optional flags below).

 INPUTS
handle - Same handle as that used when OwnCard() was called.

flags - Optional flags.

  - CARDB_REMOVEHANDLE means you want remove your
  CardHandle structure from the notification list
  whether or not you currently own the credit-card
  in the card slot.  The node structure in your
  CardHandle will be removed from the notification
  list, and ownership will be released if you were
  the owner of the card.

 NOTES
This function should only be called from a task.

  SEE ALSO
OwnCard(), resources/card.i, resources/card.h
```