Programming Guide for Opus-Two Next Generation PIC Controllers

This document is intended for technically oriented organ builders and field technicians to use as a reference when configuring musical instrument applications (pipe/electronic organs). This document does not teach programming and is not a substitute for proper education and training. This document does not teach basic computer use (file navigation, use of a command prompt, etc.) and is not a substitute for that knowledge. One-on-one training is available for anyone with a technical or programming background who wants to become more familiar with the intricacies of Opus-Two.

A software toolset is required to use any of the information contained herein. This is provided by an Opus-Two dealer. Never use a toolset provided by a third party.

A PicKit2/PicKit3 is required to load compiled files into controllers. This is not a USB to serial adapter per se and is provided by Opus-Two to dealers with the toolset. This programming cable is sometimes provided to be left with the job site - in that case, a USB thumb drive is also provided to contain the toolset and configuration files. A PDF of this document should also be included.

Introduction

The Opus-Two Integrated Control System provides an unmatched level of user programmability. This depth of control enables end users to create features and extreme customization with no software changes to the core product needed. With this extreme level of control comes the ability to inadvertently create problems (both very obvious and less obvious). One of the most common sources of problems and of inquiry regards the ordering that a config file must have entries and the contents. This manual provides a block diagram of what order various "sections" should be. By following this guideline, operations will be quickest and require the least amount of troubleshooting.

Config files for consoles and chambers are compiled using different commands, but the files are constructed exactly the same way. To understand the config process, it is first important to understand the way the hardware works.

There are two card scan processes per cycle: Input Reading and Output Writing. The config file is therefore split into two parts: Input and Output.

In a pipe chamber, the input section is quite small (if anything). In a console, the combination action really spans both parts – inputs, i.e. pistons and stops, are read and the resulting coil driving occurs when driving outputs.

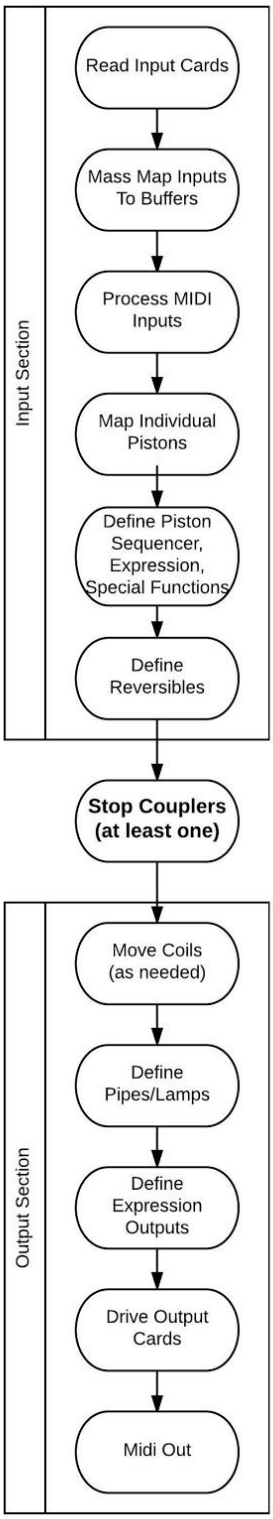
Regardless of whether the IO structure is made up of legacy/separate input cards and output cards or Next Generation combined IO cards, there are still two separate cycles. To help grasp the concept, think of the card as having completely separate input circuitry and output circuitry for the entire card. They only combine at the pin.

Input cards are read sequentially, in order. As they're being read, the data is stored in dedicated space (card in buffers). The data then needs to be mapped to where it belongs – keyboard buffers, Stop Tab buffers, piston buffers, etc.. During the output section, data is written by the config file to dedicated space (card out buffers). Once that is finished, any legacy input cards "become invisible" and the controller communicates directly with the output cards, streaming out the data defined by the config file.

IMPORTANT: The "mode switch" between input mode and output mode happens when the system sees the first coupler. As soon as it reaches that coupler, it converts input card buffers to output card buffers. Therefore you can not reference an input card pin after the first coupler, nor can you reference an output card pin before the same coupler. It also processes any pending combination action work, applies tutti/crescendo, sends chamber frames, and formats keying for pipe operations at this divide point.

The examples listed in this document are based on eb_nutley_console_1, which is distributed in the toolset.

Block Diagram/Order



- Preamble/Declarations
- If chamber file, define stop names
- Read Input Cards (console only)
- Define any reversible keyboard controls (console only)
- Map Keyboards (console only)
- Map Pistons (console only)
- Map Set/Cancel (console only)
- Map other reversible controls (console only
- Piston Sequencer (console only)
- Read and process analog inputs
- At least one stp_cplr
- Map C/A Coils to cards (console only)
- Drive any pipes or indicators
- Drive output cards
- MIDI Outputs

Opening Comments (a.k.a. things to know)

When making a configuration file, it is wise to put any unique information about the site in the opening comments. Appropriate things include church name, address, instrument size, etc. Anything that might be needed for remote troubleshooting is wise to include in the comments area. Anything that anyone else would need to know to remotely support someone at the site should be included in the comments. Comments are NOT compiled into the finished code that goes into the instrument, so there are no practical restrictions on number of lines, sentences, etc. It is far better to write too much than too little. It's often useful to even include the compile line, which prevents questions later about pin options or how to compile an older controller.

A comment in the JAL language is indicated by starting a line with two dashes "- -" or a semicolon ";".

Cards are assumed to be in numerical order from the controller card out - the closest card to the controller is number 1. When defining individual pins on an input card (either as a control button or a point of reference in code), it can be referred to in a string. For example, input card 7's 20th pin would be expressed as "in7_20". This allows any pin of any input card to be called upon into a procedure to do work. These are predefined in an external file that is called during compile automatically.

Opening configuration declarations

Header statements change things (controls, features) from their default value to the value you would like. They generally could appear anywhere in the files, but putting them in the header ensures that the changes are applied as soon as possible.

ChamberBus Send & Receive

```
Opus_Remote = low
Opus_Remotes = high
```

First line indicates that this is NOT a chamber and does not need to receive supporting data/material from other controllers to function. Second line indicates that it needs to generate said data so that other controllers can be synchronized with it and receive the data that is read here. It is important to note that if opus_remote is high, the controller will not work without being 'clocked' by receiving another controller's data. The reception of the console's data is what causes the chamber controller to execute its configuration file.

Memory mapping, combination mapping

```
comb_mem_bytes = 13
pistons_per_level = 80
combination_action = high
Opus_uSD_card ( 8 )
```

In PIC-based controllers, combination pistons and internal sequencer data is stored in an external memory module (FLASH or uSD). Having a memory map set up ahead of time assures that the system always looks in the same spot for the same thing. Adequate preparation at this stage for the size of the instrument (including any future additions) is critical to making sure organists NEVER lose combinations.

The comb_mem_bytes is determining how many bytes should be set aside for each piston on each memory level. This is important to get right the first time – a byte holds 8 bits (or stops). There is no shortage of memory – the combination actions are stored on the external uSD/FLASH card (lots of room). Make this big enough that you never have to change it again. If an organ has 35 stops, the number needs to be at least 5 (5 x 8 = 40). On a 35 stop organ with no anticipated growth, it would be appropriate to make it at least 8. This allows for storage of the state of any 'under the hood' features that are created later that require memory storage, or any stop expansions.

Also need to add a byte if planning to have the transposer settings included in combination action.

pistons_per_level is the same sort of thing as comb_mem_bytes where you never want to guess too low. A common way is to count the amount of room for pistons, so that if any are added later, space is allowed for them. These do not take up much memory, and any reformatting of either of these two variables later will make pistons above memory level 1 scramble. If an organ has 10 generals and 5 divisionals on each of 3 divisions, that is a total of 25 pistons (coupler reversables and the like do not count). If it's conceivable that the toe studs might be requested to be separate

memory or pistons could be added, they should be accounted for from the very beginning.

combination_action = high is merely activating the routines that store and process piston presses. Without this (or with it set to =low), pressing pistons will not move any stops. This mode can be helpful when trying to get a console up and running (forcing the combination action off prevents it from trying to move things before it's known that they are wired correctly). If this is off, pistons can still be seen on the screen in the pistons buffer, they just won't do anything. A newly wired console should have combinations_action = low until as much wiring has been verified as possible.

```
Opus_uSD_Card ( 8 )
```

opus_uSD_card tells the config there is a uSD card and how big the uSD option card is (leave at 8 regardless of the card size). The current toolsets auto-detect card size.

```
piston_transposing = high
```

This (when high) allows transposer settings to be stored with pistons. This setting defaults to low/off.

```
use_new_display ( high )
```

There are two types of displays that have been used on Opus-Two systems, one of which has a rather large backplane board (with built in control buttons), and the other has a thin backplane board and wiring headers for pistons. use_new_display is for the thin backplane board. The (high) determines whether the display button inputs are read from the backplane board or not.

```
Display_present = low
```

This is on/high by default. During startup, the controller will wait for the display to boot and start responding to commands. By adding this line, the controller does not attempt to communicate with the display, and startups are quicker on systems that don't have display units. As a side note, most systems should have displays for troubleshooting purposes.

```
crescendo_enable = low
```

As it appears, this line enables or disables the crescendo, including the display appearance.

```
use_reits = low
```

When high, enables a special menu to adjust reit timer values. This also enables sending those values to the chamber controllers on startup.

```
use_digital_tuning = low
```

When high, enables a special menu to enable fine tuning of Hauptwerk digital voices.

```
use_midi_selects = low
```

When high, enables a special menu to select MIDI voices for an external expander.

Special C/A formatting constraints (reversed coil wiring, 8-Block wiring, etc.)

Opus-Two can output coil data in four unique ways, and the user can further map from there. The order of the coils is fixed based upon the order of the stop-sense data in the TI blocks.

1) Alternating on-off coils, on coils first.
2) Alternating off-on coils, off coils first.
3) Full bytes of on-off coils (8 on followed by 8 off).
4) Full bytes of off-on coils (8 off followed by 8 on).

```
coils_reversed = high
```

By default, this is “low” and the combination action drives alternating coils (on first). Enabling this reverses the coils to be off-first.

```
tab_grouping = high
```

By default, this is “low.” Enabling this causes the C/A routines to output alternating bytes of on coils followed by bytes of off coils (example 3 above). Enable coils reversed with it to get example 4 above.

```
Drive_all_tabs = high
```

By default, this is “low.” Enabling this causes the C/A routines energize the proper coil on each individual tab whether the tab needs to move or not. This uses extra energy but is helpful with sticky tabs that get stuck halfway or in applications where all tabs need to be driven mechanically (such as electrifying a mechanical C/A).

```
CA_One_Pulse = high
```

By default, this is “low.” Enabling this causes the C/A routines to output a single fixed length pulse to the coils. The C/A (without this being on) will stop driving coils once a tab has been satisfactorily moved.

Home Screen/Menu Options

```
Show_mem_level = low
```

By default, this is “high.” Disabling this prevents the memory level from being displayed (useful in record/playback only systems). The up/down buttons still affect the memory level, it just isn’t shown.

```
Show_cres_step = low
```

By default, this is “high.” Be aware that turning this off also disables the on-screen indication of Sforz. Inline logic can be scripted to only show this when cresc_step > 0.

```
Use_custom_display = high
```

By default, this is “low.” This prevents the display driver from blanking the third line. Once the third line is not being blanked, the config file can print anything on the line the user desires. This is often handy for last piston pressed, status of some event, etc. When doing this, make sure to qualify your custom third line with “if m_state == 23 “ or else it will conflict with sub-menus and sub-screens.

```
Show_spinner = low
```

This disables the character spinner in the upper right corner. This should be done before turning the instrument over to the client. Leave the spinner active during troubleshooting as it indicates the controller card is still running..

```
Track_enabled = low
```

By default, this is “high.” Disabling this bit will hide the track number. Like hiding the memory level, this track number can still be changed if the correct buttons are pressed but the user will not see it.

RS Data TX/RX

The Opus-Two chamber bus carries the information from the console to a virtually unlimited number of chambers. This dataline is a balanced RS-485 serial connection, where A is positive, B is negative, and C is reference ground. Be aware that the reference ground is not firmly connected to controller negative (there is a resistor in series). As long as the header statements have listed opus_remotes = high then the controller will send chamber frames.

The frames come in several sizes. By default, the frames are 9 blocks. If the controller needs to send more (or less), it needs to be defined in the config files on each end by listing "console_blocks = #".

The normal blocks are:

1) Acc/Swell
2) Great
3) Acc2/Choir
4) Solo
5) Pedals1/Pedals2
6) TI1 (Stops 1-64)
7) TI2 (Stops 65-128)
8) TI3 (Stops 129-192)
9) Expressions
10) Great2
11) Stentor
12) TI4 (Stops 193-256)
13) TI5 (Stops 257-320)
14) TI6 (Stops 321-384)
15) TI7
16) TI8
17) TI9
18) TI10
19) TI11
20) TI12
21) TI13
22) TI14
23) Exprsb1
24) Exprsb2

Stop Name Declarations (Chamber Only)

```
Var Bit Pd_Bourdon_16    is Stop_1
Var Bit Pd_Diapason_8    is Stop_2
Var Bit Pd_Flute_8       is Stop_3
```

Opus-Two allows alternative names to be assigned to each bit. This gives the user the ability to name each bit to match the stop control. This makes configuration, troubleshooting, and documentation easier. In code examples throughout this document, stops are referred to by name. These names get defined in the beginning of the file. They are not case sensitive. Use underscores instead of spaces.

The stops declared in the chamber are the same stops sent from the console. The cmn-mem diagnostic tools on the console can be used to confirm which stop is assigned to which number.

Read Input Cards

```
Read_input_cards ( 15 )
```

This tell Opus-Two how many cards to read input from. Read all IO cards, even if you don't think you need data from them.

Invert Input Card

```
Invert_input_card ( 3 )
```

Most IO cards used in Opus-Two are positive polarity. If a negative polarity card is used, this procedure will invert the data from the card to read correctly.

Mapping Input Cards

Once the input cards have been read into the system buffers, the data needs to be distributed to the correct place within the system. This is where order is put to the data that comes in.

The following procedures use "short names" for destination buffers. The following buffers are legal/valid names to use with these procedures:

Swell / Acc	Great2	Dwr_state
Great	Bombrd	Pistons
Solo	TI1 .. TI14	Pistons_1
Choir / Acc2	Exprs1	Cresc_in
Pedals	Exprs2	Temp_w1 .. Temp_w4
Expr	Temp_wrk	

Map (card , pin , number of bits , destination buffer)

```
Map ( 2 , 1 , 61 , Swell )
```

One of the most efficient (but least flexible) procedures, the map allows a chunk of data to be moved from input buffers (starting at a specific card/pin) to a destination buffer. The data is pasted in the destination buffer starting at the first bit (the insert point can't be changed).

Map_Fragment_Merge

This procedure has been replaced with map_in.

Map_In (card , pin , number of bits , destination , starting bit in destination)

```
Map_In ( 2 , 1 , 61 , Swell , 1 )
```

This procedure replaces map_fragment_merge. Not as efficient as map, but much more flexible. This allows a chunk of data to be moved from input buffers (starting at a specific card/pin) to a destination buffer. The destination insert point can be defined.

Bit_Map (byte , bit 1 , bit 2 , bit 3 , bit 4 , bit 5 , bit 6 , bit 7 , bit 8)

```
Bit_Map ( pistonsb1 , in1_10 , in1_11 , in1_12 , in1_13 ,
                      In1_22 , in1_14 , in1_15 , in1_61 )
```

The bit map allows any byte to be custom built from predefined bits within the config file. This is one way to assemble pistons in the order needed or even keys if they are really out of order.

Important Note About Piston Order

General Pistons 1-10 (if they exist) should always be mapped to piston buffer 1-10. The “quick dial” feature where the organist holds general cancel and “dials” their desired memory level on the generals relies on this placement.

Including any rollers for expression/controllers

While not common, some consoles still have contact rollers (for various reasons) instead of analog inputs. Each expression byte has a value range from 0-255 (as sent to the chamber), and if being sent via MIDI, 0-127. Because it is unlikely that a roller actually has 127 stages and because it is likely desirable to use the full 127 bit range, procedures exist to convert contact rollers to expression values. It is, however, just as simple to decide how many bits each contact is worth and simply write a freehand if statement. Numerous examples exist in libraries.

MIDI Inputs

When MIDI is received by an Opus-Two controller, if it is a note, it is processed by a background handler. This handler goes to special/reserved memory space (64 bits x 16 channels) and ticks the status of the appropriate note either on or off, depending on the message. This "local record" of the status of MIDI notes is maintained and assumed accurate unless cleared by commands in the config file.

Other MIDI traffic (SYSEX, NRPN, RPN, Controller, Expression, etc.) are placed into a MIDI receive buffer. It is up to the config file to retrieve these messages and do something with them (or else they will be discarded).

Lowest note value

```
Uniform_midi_map = low
```

The uniform midi map bit tells the background note processing routines where to start decoding notes. Each midi channel has a possibility of 128 different/unique note messages, but keyboards are only 61 notes (and Opus-Two keying buffers in the console are only 64 bits), so the decoding has to have a window within those 128 notes that it works.

	Uniform_midi_map = high	uniform_midi_map = low
Channel 0-5 (1-6)	Starts at note 36	Starts at note 0
Channels 6-15 (7-16)	Starts at note 0	Starts at note 0

Decoding and Processing Note Messages

```
map_midi_in ( 1  , 61 , Swell  )  -- Swell Keyboard Sequencer In
map_midi_in ( 2  , 61 , Great  )  -- Great Keyboard Sequencer In
map_midi_in ( 4  , 32 , Pedals )  -- Pedal Keyboard Sequencer In
map_midi_in ( 10 , 64 , TI1    )  -- stops 1  to 64
```

In this particular case, the console is receiving 61 Swell keys on channel 1, 61 great keys on channel 2, and 32 pedal keys on channel 4. The start points are predefined within the channel (note 36). 64 stops are being received as note messages on channel 10, starting at note 0.

Controller/Expression/Any 3 Byte Message

Any and all non-note processing needs to be done in a dedicated loop within the config file, like this:

```
while midi_decode_work loop
   Midi_Bn_ctrl_decode ( 0xB0 , 7  , midi_4_vol )
   Midi_Bn_ctrl_decode ( 0xB1 , 7  , midi_5_vol )
   discard_unused_midi
 end loop
```

Each config file should only have one of these loops, and anything non-note gets decoded inside of it. The Midi_Bn_ctrl_decode procedure looks through the MIDI buffer and tries to match the first two bytes, and then passes the next byte back out to where you specify. So in the first example, if there was a 3 byte message that had been received that was 0xB0/0x07/0x7F, the system would match the first two bytes, and then it would output the 0x7F to midi_4_vol. If it doesn't match the first two bytes, it doesn't touch/change the third byte.

All midi_decode_work loops must have discard_unused_midi at the end or else the loop may not complete.

Custom message protocols

While we encourage users to work within notes and standard 3 byte messages, any protocol can be custom written rather easily. See jb_lehighton_1_allen.jal for examples. That site decodes Allen piston messages and uses them to fire an internal C/A, remaining synchronized with Allen memory level. Allen uses 2 byte messages to send that data, so custom MIDI drivers were written.

It is also worth noting that once MIDI messages are received into a buffer, they can be copied anywhere from that buffer in the config file.

Combination Action Items

Set, Cancel, and Sforzando

```
Set_piston ( in2_12 )
Cancel_piston ( in2_13 )
sforz_button ( in2_15 , 1 )
```

The sforz button second parameter is sforzando number. Current toolsets support values of 1-4.

Stop Reversables (Single-Tab Only)

A reversible control is one that toggles the state of a stop. There are two schools of thought regarding these, one requires a bit of intelligence and one is a dumb control.

```
Reversable_tab ( in3_2 , 28 )
```

The “dumb control” simply turns the tab on if it’s off and off if it’s on. It simply reverses its state every time the button is pressed. The example above toggles stop 28 every time in3_2 is pressed.

Stop Reversables (East Coast Style)

An east coast reversable group is a section of tabs that work together (such as Swell to Great 16, 8, and 4). This procedure checks if any of them are on, and if so turns them off. If none of them are on, the first defined stop (typically the 8’) is turned on (only).

```
east_coast_Reversable ( in3_2 , 28 , 27 , 29 , 0 )
```

Pushbutton Stops (Reversible Stops)

The same concept of a reversible tab may be desired without actually having something move (turning a memory bit on and off). Sometimes called “lit reversables” even though no light is needed for the function, these are turned on and off by simply pressing the button. They are ‘bound’ to a stop number and toggle that stop number on and off. They also have a separate bit they toggle in the “dwr_stop” buffer. This bit is preserved from pass to pass, whereas the stop bit is not. During combination action setting (when a piston is pressed), the procedure will use the status of the stop being set by the C/A to determine the future state of the lit reversible. This allows them to be masked into combination action memory per piston.

```
tng_Blind_Reversable_Stop_a ( In5_16 , 240 , 5 )
```

This example uses a button wired to in5_16 to toggle stop 240 on and off. Drawer Stop #5 is used to preserve the state from pass to pass.

A second/additional procedure exists:

```
tng_Blind_Reversable_Stop_b ( In5_16 , 10 , 5 )
```

Because the PIC environment is limited to 8 bit integers, the stop number can only be up to 256. This adequately allows the user to configure any stop in the first 4 TI blocks (64 stops per block), and the “b” version of the procedure allows configuration in the next 4 TI blocks. So the stop number in the “b” version is added to 256 to determine the real stop number. In other words, a value of 1 in the “b” procedure is actually stop 257.

Pushbutton Stops (Reversible Stops)

A similar procedure exists that is completely separate from the combination action. These reversable buttons can not be set into the C/A and therefore can not be cleared by a piston.

```
--                              button  dwr#
   tng_Blind_Reversable_Btn ( In10_5 , 1 )
```

Analog Inputs (from expression shoes)

```
exprb1 = fetch_adc_val ( 1 )
```

The fetch_adc_val returns the current value of the analog number. This number directly corresponds to the number silkscreened on the controller card. The example above copies analog 1 directly to exprb1. To make this instantly compatible with MIDI (no values greater than 127), simply divide by 2 like this:

```
exprb1 = ( fetch_adc_val ( 1 ) / 2 )
```

Sometimes analog inputs can drift. These drifts can be quite annoying and can cause swell blades to flutter. If a swell blade is set to open if the value is over 100, and the value is fluctuating between 100 and 101, the blade will flutter with the value. There are two approaches to this: one is to have a minimum change required in the analog value before the change is accepted, and the other is to filter the value. The filter uses an averaging scheme to average the last x number of passes.

```
exprb1 = variable_filter_sample ( 3 , fetch_adc_val ( 1 ) , saved_misc )
```

This example will average the last 3 passes (storing the running average in saved_misc). Legal values for the number of passes are 1-5.

A fully processed analog input is possible using filter_analog input:

```
--                          threshold   a#  min   range  store byte
exprb1 = filter_analog_input (   5    ,  1  , 10 , 128 , midi_15_vol )
```

Threshold is how far the value has to drift before a new value is passed into the file. A# is analog number (as screen printed on the card), min is how much to scrub from the bottom of the shoe, range is a manipulator. A value of 128 means no changes, larger values are multipliers, and smaller values are divisors. A value of 127 will reduce the value, a value of 129 will enlarge it. Use values here as necessary to get the range of shoe values desired. A store byte keeps track of threshold changes.

MIDI Expression Merging (if necessary)

Depending on the use of MIDI in a specific console, it may be necessary to merge together incoming expression values with the values already scanned (from the shoes). Opus-Two makes no assumptions about how this should be handled. Opus-Two also makes no assumptions about how to age the incoming value. How the shoe is treated when a MIDI messages is received and how long that treatment occurs are entirely up to the config file to manage.

The most common way to manage this is to have a timer running just for MIDI merging.

```
while midi_decode_work loop
  Midi_Bn_ctrl_decode ( 0xB0 , 7  , midi_4_vol )
  Midi_Bn_ctrl_decode ( 0xB1 , 7  , midi_5_vol )
  Midi_Bn_ctrl_decode ( 0xB2 , 7  , midi_6_vol )
  if _did_midi_work then saved_misc = 255 end if
  discard_unused_midi
end loop
if ( ( midi_vol_timer < 5 ) | ( ! welcome_off ) ) then
  midi_4_vol = 255 midi_5_vol = 255 midi_6_vol = 255
end if
if midi_vol_timer > 0 then
  midi_vol_timer = midi_vol_timer - 1
end if
```

The timer value check and the high value changes below the loop are to make the saved bytes to a known value. If they are 255 (which isn't even a legal MIDI value) we can check for that later and know which ones have actually gotten valid replacement data. This way, only current valid replacement values are used, the rest are still under the control of their expression shoes.

This loop is typically closer to the top of the config file (with all the other MIDI In activity). Further down in the file (after analog processing), some code will actually merge the MIDI value:

```
    if midi_4_vol < 128 then exprb1 = midi_4_vol end if
    if midi_5_vol < 128 then exprb2 = midi_5_vol end if
    if midi_6_vol < 128 then exprb3 = midi_6_vol end if
```

Couplers (always have one!)

```
Stp_cplr ( stop_23 , Sw_k , Gr_k , p_8 )
```

This example will copy great keying to the swell keying buffers when stop_23 is on.

Most consoles don't have any couplers because they don't have any pipe outputs. If MIDI outputs were to be coupled, this would accomplish that, but typically that is undesired. But a stp_cplr is always needed, so one can be put in that is forced off:

```
Stp_cplr ( low , Sw_k , Sw_k , p_8 )
```

The "low" bit being passed in prevents this from ever turning on. But it's presence in the configuration file engages all the under the hood procedures (chamber data send, combination action, crescendo, tutti, etc.).

{ Actually, C/A could operate without the stp_cplr - but the output buffers not cleared or set up right, so could get weird results on buffer copy, move or merge. This stp_cplr does the record / playback operations as well as chamber data }

Drive C/A Coils

When a piston is pressed, if combination_action = high and if a range/mask has been set for the piston, when the first stop/coupler is reached, the C/A will scan through the piston buffers. If it finds a piston on, it will try to process it.

The C/A procedures take the existing TI buffers (state of tabs) and compare them to the desired state (piston contents in masked area + existing state in unmasked area) and creates a "next state" which is what the tabs need to look like. From there, the procedures compare the existing state against the next state and decide whether an on coil needs to fire, an off coil needs to fire, or neither. Two buffers are created from this (on_coils and off_coils). These buffers are then merged together into the TCO (tab coil out) buffers following the rules declared in the config header (coils reversed, tab grouping, etc.).

TCO blocks (of 64) have the coil data for 32 tabs per TCO block.

The easiest/most common way to handle stop coils is to copy the TCO blocks (en mass) to the output cards, like this:

```
if comb_coils_enabled then
   move_buf ( TCO1 >> 8    , TCO1    , Out_1 >> 8 , Out_1 , 27 )
 end if
```

This example moves 27 bytes of coil data to output buffers, starting at card 1, pin 1. That is 3 cards + 3 headers on the 4th card worth of coil data, which is enough to move 108 stop tabs.

Once the coils are copied over to the output buffers, any data manipulation necessary to correct for wiring can occur.

Drive Lamp Outputs

```
stp_ctrl ( Sforz_1         , 10 , 49 )
stp_ctrl ( Cresc_step > 0  , 10 , 50 )
stp_ctrl ( welcome_off     , 10 , 51 )
```

In the examples, a sforzando lamp, crescendo active lamp, and power lamp are being driven to card 10, pins 49, 50, and 51.

Drive Pipes

All pipe outputs in Opus-Two are treated the same; like unit ranks. Station or Primary chests are treated as a unit rank with a single stop. Various output wiring styles are supported, including a pin-by-pin definition system, allowing completely non-patterned wiring to be supported.

```
Stp_seg     ( Sw_Flautino_2      , p_2  , Sw_k )
Stp_seg     ( Sw_Flute_4         , p_4  , Sw_k )
Stp_seg     ( Pd_Flute_4         , p_4  , Pd_k )
Stp_seg     ( Sw_Gedeckt_8       , p_8  , Sw_k )
Stp_seg     ( Pd_Gedeckt_8       , p_8  , Pd_k )
Stp_seg     ( Pd_Lieblich_16     , p_16 , Pd_k )
Stp_seg     ( Gr_Gedeckt_16      , p_16 , Gr_k )
Stp_seg     ( Sw_Gedeckt_16      , p_16 , Sw_k )
rank_trim   ( 97 , P_16 )
rnk_seg_out ( 97 , 7 , 1 , fmt_c_a )
```

In this example, a number of stops have been defined to play from a unit flute. Of importance is the descending order of the stp_seg statements; they are in order from highest pitch to lowest pitch. The rank trim defines the number of pipes in the rank and the lowest pitch pipe available. In this example, all 97 notes are located in order, wired from card 7 pin 1 onward.

```
Chest_Enable = ( Ch_Diapason_8 | Ch_Flute_O_8 | Ch_Dulciana_8 )
Stp_seg     ( chest_enable , p_8  , Ch_k )
rank_trim   ( 61 , P_8  )
rnk_seg_out ( 61 , 8 , 10 , fmt_c_a )
```

This example is of a primary chest, where 3 stops play from the same chest. The bit “Chest_Enable” is predefined and can be bound to a large logical OR group - so if any of those stops are on, chest enable is also on. That then enables the stp_seg line, and makes the primary play.

A primary-style chest has two components to it, notes and stop actions. To make the stop actions work, simply use stp_ctrl statements, such as these:

```
stp_ctrl ( Ch_Diapason_8   , 9 , 7  )
stp_ctrl ( Ch_Diapason_8   , 9 , 8  )
stp_ctrl ( Ch_Flute_O_8    , 9 , 9  )
stp_ctrl ( Ch_Flute_O_8    , 9 , 10 )
stp_ctrl ( Ch_Dulciana_8   , 9 , 21 )
```

In this case (an Austin), the Diapason and large flute have two separate stop actions. This is not a problem, simply define them twice and point to the correct pins (in the case of the diapason, card 9, pins 7 and 8).

Sometimes wiring is not in order, but just seems to be randomized for whatever reason. In that situation, this can be done:

```
Stp_seg     ( Pd_Bourdon_16 , p_16  , Pd_k )
rank_trim   ( 32 , P_16  )
wb_7_note_adjust
    note_out ( 5 , 39 ) -- 1
    note_out ( 5 , 21 ) -- 2
    note_out ( 5 , 37 ) -- 3
    note_out ( 5 , 23 ) -- 4
    note_out ( 5 , 35 ) -- 5
    note_out ( 5 , 25 ) -- 6
    note_out ( 5 , 33 ) -- 7
    note_out ( 5 , 27 ) -- 8
    note_out ( 5 , 26 ) -- 9
    note_out ( 5 , 29 ) -- 10
    note_out ( 5 , 30 ) -- 11
    note_out ( 5 , 31 ) -- 12
    note_out ( 5 , 32 ) -- 13
    note_out ( 5 , 24 ) -- 14
    note_out ( 5 , 34 ) -- 15
    note_out ( 5 , 28 ) -- 16
    note_out ( 5 , 36 ) -- 17
    note_out ( 5 , 22 ) -- 18
    note_out ( 5 , 38 ) -- 19
    note_out ( 5 , 20 ) -- 20
    note_out ( 5 , 40 ) -- 21
    note_out ( 5 , 41 ) -- 22
    note_out ( 5 , 42 ) -- 23
    note_out ( 5 , 43 ) -- 24
    note_out ( 5 , 44 ) -- 25
    note_out ( 5 , 45 ) -- 26
    note_out ( 5 , 46 ) -- 27
    note_out ( 5 , 47 ) -- 28
    note_out ( 5 , 48 ) -- 29
    note_out ( 5 , 49 ) -- 30
    note_out ( 5 , 50 ) -- 31
    note_out ( 5 , 51 ) -- 32
    wb_7_note_adjust
```

Every single note is defined, in order. In the example above, comments are used to keep track of what note numbers are which line. This makes troubleshooting, documenting, and changes much easier.

“wb_7_note_adjust” must be run before the note outs or any procedure that uses note outs (typically indicated in procedure name).

Sometimes patterns are easy to see and can be defined in a loop, such as this:

```
  var byte jj = 0

  Stp_seg     ( Sw_Vox_Humana_8        , p_8  , Sw_k )
  rank_trim   ( 61  , P_8  )
  wb_7_note_adjust
  for 16 loop
    note_out ( 10 , 16 - jj )
    note_out (  5 , 19 + jj )
    jj = jj + 1
  end loop -- 1 - 32

  jj = 0
  for 12 loop
    note_out ( 12 , 33 + jj )   note_out ( 12 , 61 - jj )   jj = jj + 1
  end loop -- 33 - 56
  jj = 0
  note_out ( 12 , 46 ) -- 57
  note_out ( 12 , 49 ) -- 58
  note_out ( 12 , 45 ) -- 59
  note_out ( 12 , 47 ) -- 60
  note_out ( 12 , 48 ) -- 61
wb_7_note_adjust
```

Several procedures are predefined to allow these unique wiring patterns to be easily built:

```
  Stp_seg     ( Sw_Vox_Humana_8        , p_8  , Sw_k )
  rank_trim   ( 61  , P_8  )
  inc_rnk_seg ( 12 , 6 , 1 ) -- 12 notes starting card 6 pin 1
  dec_rnk_seg ( 12 , 7 , 12 ) -- 12 notes descending start @ crd 7 pin 1
  tn2_rnk_seg ( 19 , 2 , 1 , fmt_d_a , 3 , 10 , fmt_d_d )
```

“inc_rnk_seg” starts at the card/pin and pastes notes up the card (pin 1 then 2 then 3, etc.).
“dec_rnk_seg” starts at the card/pin and pastes notes downward (pin 10 then 9 then 8, etc.).

“tn2_rnk_seg” allows diatonic definitions to be created. The number of pairs to be distributed is defined first (19 pairs = 38 notes). Next is the card/pin for the first bit of each pair, followed by the format. “fmt_d_a” uses a inc_rnk_seg type distribution and counts up. “fmt_d_d” uses dec_rnk_seg and counts down. These can be mixed in any of the 4 possible combinations.

All of these procedures automatically add wb_7_note_adjust and can cross card boundaries.

Stp_seg_l

This chamber procedure works like any other stp_seg, but allows you to specify a specific part of the keyboard to be copied over. There are two numbers after the standard definition. The first number is an offset value, the second number is the number of notes to copy over. Values of "0 , 12" would copy notes 1-12 only. Values "12 , 24" would copy notes 13-36. Values of "0 , 61" would behave the same way as a normal stp_seg. Remember, the values of those digits affect the keyboard keys that are copied. The defined pitch (like any other stp_seg) defines where those notes are pasted. It is therefore possible to extract an octave from the keyboard and paste it an octave lower, two octaves higher, two octaves and a fifth higher, etc.

Resultants

Thanks to the extreme flexibility of Opus-Two, there are many ways to accomplish resultants (and many ways the user may want them to work). An example:

```
Stp_seg     ( Sw_Flautino_2      , p_2  , Sw_k )
Stp_seg     ( Sw_Flute_4         , p_4  , Sw_k )
Stp_seg     ( Pd_Flute_4         , p_4  , Pd_k )
Stp_seg     ( Sw_Gedeckt_8       , p_8  , Sw_k )
Stp_seg     ( Pd_Gedeckt_8       , p_8  , Pd_k )
Stp_seg_l   ( Pd_Resultant_32    , p_10_2f3 , Pd_k , 0 , 12 )
Stp_seg     ( Pd_Lieblich_16     , p_16 , Pd_k )
Stp_seg     ( Gr_Gedeckt_16      , p_16 , Gr_k )
Stp_seg     ( Sw_Gedeckt_16      , p_16 , Sw_k )
Stp_seg_l   ( Pd_Resultant_32    , p_16 , Pd_k , 0 , 12 )
Stp_seg     ( Pd_Resultant_32    , p_32 , Pd_k )
rank_trim   ( 97 , P_16 )
rnk_seg_out ( 97 , 7 , 1 , fmt_c_a )
```

In this example, a resultant 32 tab plays an open fifth for the first octave, and then starts over at low C of the 16' for note 13. This in effect plays a resultant for the bottom 12 and plays at 32' pitch for the rest of the pedal board. The bottom stp_seg (32') calls for notes that the rank doesn't support (a 16' rank has no bottom 32' octave). These notes are trimmed off and not played.

Custom Pitches And Couplers

Every pitch is equivalent to a numeric value that represents how far apart different pitches are (relatively). If 32' pitch is a value of 1, then 16' pitch would be a value of 13, 8' pitch a value of 25, etc. This numeric based flexibility means that every conceivable pitch can be defined for a stp_seg, stp_cplr, or even a rank_trim. In the event that a custom pitch is desired that hasn't been defined, the appropriate value can be inserted in place of the p_ value. This is the list of pre-defined pitches and their relative offset values:

p_0	= 0	p_2_2f3	= 44	p_2f7	= 83
p_32	= 1	p_2_2f7	= 47	p_1f4	= 85
p_21_1f3	= 8	p_2	= 49	p_2f9	= 87
p_16	= 13	p_1_7f9	= 51	p_1f5	= 89
p_13_4f9	= 16	p_1_3f5	= 53	p_2f11	= 90
p_12_4f5	= 17	p_1_5f11	= 54	p_1f6	= 92
p_12_7f11	= 17	p_1_1f3	= 56	p_1f7	= 95
p_11_3f10	= 18	p_1_1f7	= 59	p_1f8	= 97
p_10_2f3	= 20	p_1	= 61	p_1f9	= 99
p_9_1f7	= 23	p_8f9	= 63	p_1f10	= 101
p_8	= 25	p_4f5	= 65	p_1f11	= 102
p_6_2f5	= 29	p_8f11	= 66	p_1f12	= 104
p_5_1f3	= 32	p_2f3	= 68	p_1f14	= 107
p_4_4f7	= 35	p_4f7	= 71	p_1f16	= 109
p_4	= 37	p_1f2	= 73	p_1f18	= 111
p_3_5f9	= 39	p_4f9	= 75	p_1f20	= 113
p_3_1f5	= 41	p_2f5	= 77	p_1f22	= 114
p_2_10f11	= 42	p_4f11	= 78	p_1f24	= 116
p_2_3f5	= 43	p_1f3	= 80	p_1f28	= 119
				p_1f32	= 121

Coupled Keying vs Uncoupled Keying

All examples in the config file so far have referenced _k keying buffers (such as sw_k). This uses the keying buffer that is affected by couplers. There are situations where uncoupled keying is preferred. This keying is available by changing the name to have a _uk designation (such as sw_uk). The list of keying buffers:

Ac_k/Sw_k	Ac_uk	Pd_k	Pd_uk
Gr_k/Gt_k	Gr_uk	P2_k	P2_uk
So_k	So_uk	Bm_k/Et_k	Bm_uk/Et_uk
A2_k/Ch_k	A2_uk/Ch_uk	Po_k/St_k	Po_uk/St_uk

Card Output Statements

```
drive_output_cards ( 22 )
```

This tell Opus-Two how many cards are plugged into the chain and need to have data sent to them.

MIDI Out

```
Midi_out (  pd_k   , 1 , 36 )   -- Pedal Keying for Sequencing
Midi_out (  ch_k   , 2 , 36 )   -- Choir Keying for Sequencing
Midi_out (  gt_k   , 3 , 36 )   -- Great Keying for Sequencing
Midi_out (  sw_k   , 4 , 36 )   -- Swell Keying for Sequencing
Midi_out (  so_k   , 5 , 36 )   -- Solo  Keying for Sequencing
midi_ch_coder_acorn ( TI1 , ( 13 - 1 ) , 36 , high )
midi_ch_coder_acorn ( TI2 , ( 14 - 1 ) , 36 , high )
midi_ch_coder_acorn ( TI3 , ( 15 - 1 ) , 36 , high )
midi_ch_coder_acorn ( TI4 , ( 16 - 1 ) , 36 , high )
direct_midi_expression ( exprb1 , 1 , midi_1_vol )
direct_midi_expression ( exprb2 , 2 , midi_2_vol )
direct_midi_expression ( exprb3 , 3 , midi_3_vol )
direct_midi_expression ( exprb4 , 4 , midi_4_vol )
```

This block of code tells Opus-Two what to output on what channel.

The Midi_out procedure works for any defined keyboard (but not stops or expressions).

The midi_ch_coder_acorn works on any key buffer or TI buffer or expr buffer. Use caution sending an expr buffer to a MIDI channel unless it is raw roller data - that can generate an excess of MIDI traffic that won't necessarily make any sense.

direct_midi_epxression sends any byte on the listed midi channel. Last value sent is stored in the third parameter so that it is only sent when needed.

The Deep End of the Pool

Pedal Divide

There are multiple moving parts to Pedal Divide:

1) Determine whether the pedal divide will be manually set, auto set, or both.
    a) If manual, if the set button is pressed, check the pedal board for a set bit.
    b) If auto, execute the auto procedure all the time.
    c) If both, do manual for set button, but have a dwr stop for auto override.
2) An unused stops byte has to be allocated to hold the divide point in C/A memory.
3) That byte has to get bound to pd_div_point during comb_setting_cycle.
4) When the feature is in use, the divide point should be displayed on the screen.
5) Pedal Keying/Processing (the actual divide) needs to occur.
6) Configure chamber “to pedal” couplers to use P2_k instead of pd_k.

Note that the pedal divide will not be “masked” into any pistons unless it is masked in by hand during piston ranging. To mask in the pedal divide, force a divide position of top F#, and then set the ranges/masks.

An example:

```
tng_Blind_Reversable_Stop_a ( In7_16 , 240 , 5 ) -- manual div
tng_Blind_Reversable_Stop_a ( In7_17 , 241 , 6 ) -- auto div

auto_pd_div ( stop_241 , midi_5_vol , pd_div_point )

if comb_setting_cycle then
  pd_div_point = stopsb16
else
  stopsb16 = pd_div_point
end if

if ( stop_240 & in7_18 ) then -- manual + div set
  if low_bit_check ( Pedals >> 8 , Pedals , 4 ) < 32 then
    pd_div_point = low_bit_check ( Pedals >> 8 , Pedals , 4 )
  end if
end if

if ( ( ! Stop_240 ) & ( ! Stop_241 ) ) then -- if off
  show_pd_div = low
  pedalsb5 = pedalsb1
  pedalsb6 = pedalsb2
  pedalsb7 = pedalsb3
  pedalsb8 = pedalsb4
else -- if on
  show_pd_div = high
  if ( pd_div_point > 32 ) then pd_div_point = 12  end if
  if ( pd_div_point < 2 )  then pd_div_point = 12  end if

  clr_wb ( wb_6 , 4 )              -- workspace
  fill_1 ( wb_6 , pd_div_point )   -- make a mask

  pedalsb5 = pedalsb1 & ! wb_6_1   -- upper divide for coupling
  pedalsb6 = pedalsb2 & ! wb_6_2
  pedalsb7 = pedalsb3 & ! wb_6_3
  pedalsb8 = pedalsb4 & ! wb_6_4

  pedalsb1 = pedalsb1 &   wb_6_1   -- reduced pedal notes
  pedalsb2 = pedalsb2 &   wb_6_2
  pedalsb3 = pedalsb3 &   wb_6_3
  pedalsb4 = pedalsb4 &   wb_6_4
end if
```

Piston Sequencer

```
--                        enable     next          previous    top gen  holder
tng_piston_sequencer ( high , in7_8 | in8_23 , in7_7 | in8_22 , 10 , saved_solo )
```

If the enable bit is on (should either be “high” for always on or a drawer bit for the user to enable/disable), the tng piston sequencer will keep track of the last general pressed. This is accomplished by checking for pistons with a value less than the “top gen” (10 in the example). This value is stored in the holder value (saved_solo in the example). By keeping the general separate from the background last_piston_pressed, it is able to properly sequence even when divisional pistons have been pressed. The holder value can be referenced anywhere in the file where the last general is helpful (including displays). Buttons can be fed to the next and previous parameters, and can have logical OR separators for multiple buttons.

Input Cards in Chambers

```
chamber_start_card_input
read_input_cards ( 16 ) -- read them all , why not
map_in ( 1 , 1 , 61 , Solo , 1 ) -- for tuning
map_in ( 1 , 62 , 3 , TI4  , 1 ) -- stop controls
```

It is important when reading input cards in the pipe chamber that the “chamber_start_card_input” is placed. This prevents the chamber from thinking it’s a console with input cards.

Anything that gets mapped into the same place as data from the console should use a merge procedure. The example above uses an unused keying buffer, so merging is not necessary.

This should be placed below the bit declarations for stop names but before any stop couplers.

Custom Display Contents

One of the most common display customizations is to show the last piston pressed on the screen. In order to customize the third line, you have to declare use_custom_display = high at the top of the config file. This prevents the display driver from wiping/blanking the third line.

```
if m_state == 23 then -- home screen
 VFD_blank_line ( 3 , 20 )
 go_xy ( 3 , 0 )
 if last_piston_pressed < 30 then
   VFD = "L"
   VFD = "a"
   VFD = "s"
   VFD = "t"
   VFD = " "
   VFD = "P"
   VFD = "i"
   VFD = "s"
   VFD = "t"
   VFD = "o"
   VFD = "n"
   VFD = " "
 end if
 if last_piston_pressed < 10 then
   VFD = "G"
   VFD = "e"
   VFD = "n"
   VFD = " "
   VFD_decimal ( last_piston_pressed + 1 , 2 )
 elsif last_piston_pressed < 15 then
   VFD = "S"
   VFD = "w"
   VFD = " "
   VFD_decimal ( last_piston_pressed - 9 , 1 )
 elsif last_piston_pressed < 20 then
   VFD = "G"
   VFD = "r"
   VFD = " "
   VFD_decimal ( last_piston_pressed - 14 , 1 )
 elsif last_piston_pressed < 25 then
   VFD = "C"
   VFD = "h"
   VFD = " "
   VFD_decimal ( last_piston_pressed - 19 , 1 )
 elsif last_piston_pressed < 30 then
   VFD = "P"
   VFD = "d"
   VFD = " "
   VFD_decimal ( last_piston_pressed - 24 , 1 )
 end if
end if
```

Auto-Bass (Console Based)

The console based-auto bass trick coupler takes the lowest held note below the threshold value and plays it in the pedalboard. This auto-bass coupler will play the lowest note possible (if B2 is played in the keyboard, low B will be played in the pedalboard).

```
    Console_Auto_Bass ( dwr_4 , 25 , Great >> 8 , Great )
```

It's also possible to have the threshold not hard coded. Instead of the hard coded note 25 above, it could be a variable instead:

```
--                      trigger  threshold         key buffer
    Console_Auto_Bass ( dwr_4 , saved_misc , Great >> 8 , Great )
```

Any custom code can be implemented to change the value, but the easiest way is like this:

```
    set_a_split_point ( in6_24 , saved_misc , Great >> 8 , Great )
```

This will set the value of the highest note value to saved_misc.

Saving a byte value in C/A

Lots of byte values would be helpful to store in the C/A. Pedal divide points, manual divide split points, threshold values, etc. would all be helpful to store on pistons.

To maintain a value in the C/A for recall on a general piston, this procedure can be used:

```
store_and_recall_byte ( low , Stopsb16 , saved_misc )
```

The first bit (set low in the example) sets all bits of the second byte high. This should be used/on during range/mask setting. Once the ranges and masks are set, this bit can be set low.

To automate this a little more, if the console has a range button, this could be tied to that with some conditional logic checking. If the console has a range button on card 3 pin 55 and 10 general pistons, an example could be:

```
var bit ranging_bit = low
if ( ( in3_55 ) & ( low_bit_check ( Pistons >> 8 , Pistons , 8 ) < 10 ) ) then
  ranging_bit = high
end if
store_and_recall_byte ( ranging_bit , Stopsb16 , saved_misc )
```

Sostenuto

Sostenuto captures all the notes held when the feature is engaged (and only those notes) and allows them to continue to sound. Sostenuto can be configured in the console or the chamber. The obvious advantage of configuring in the console is that it can be customized in one nearby place and affect the entire instrument. Moving the feature to the chamber allows customization in each controller cards implementation (which may or may not be useful).

An example of a sostenuto that can optionally include pedal notes in the capture:

```
if dwr_11 then            -- feature enabled
  if in4_26 then          -- kickswitch/control
    if sostenuto_on then  -- don't accept new notes
      merge_buf ( temp_w2 >> 8 , temp_w2  , Great >> 8  , Great  , 8 )
      if dwr_12 then
        merge_buf ( temp_w1 >> 8 , temp_w1  , Pedals >> 8 , Pedals , 8 )
      end if
    else  -- do a new initial capture
      sostenuto_on = high
      move_buf ( Great >> 8  , Great  , temp_w2 >> 8 , temp_w2  , 8 )
      move_buf ( Pedals >> 8 , Pedals , temp_w1 >> 8 , temp_w1  , 8 )
    end if
  end if
else -- not active
  if sostenuto_on then  -- release notes by clearing buffer
    zero_buf ( temp_w1 >> 8 , temp_w1  , 8 )
    zero_buf ( temp_w2 >> 8 , temp_w2  , 8 )
    sostenuto_on = low
  end if
end if
```

This code snippet is taken from jb_sbto_7.

In consoles, the temp_w1 and temp_w2 buffers are preserved from pass to pass and are perfect for an application like this.

Dwr_12 includes the captured pedal note. Note that the pedal note is always included in the capture, but only included in ‘playback’ if requested.

As all of these things in the deep end - these are guidelines and can be customized to any site. It is important to understand the contents of the procedure before implementing it. It is equally important to make sure that the correct variables are being called and that buffers being used aren’t used for something else!

In this particular procedure, it is important to know that sostenuto_on is a bit declared outside the config file - it is there for you to use. Dwr_11, dwr_12, and in4_26 are just examples. Any drawer bit can be used.

Legato

Legato captures all notes engaged when the feature is turned on and continues to capture notes as they are pressed. Legato can be configured in the console or the chamber. The obvious advantage of configuring in the console is that it can be customized in one (nearby) place and affect the entire instrument. Moving the feature to the chamber allows customization in each controller cards implementation (which may or may not be useful).

An example of a console legato:

```
console_legato_kyb ( in3_26 , so_k , so_k )
```

Despite the fact the key names are “post processing” names, the buffers affected are pre-processing.

Sostelegato

Sostelegato engages an semi-intelligent capture mode within a console. The sostelegato captures everything pressed at once. Once all keys are released, the notes will continue to sound until something new is played. At that point, the sostelegato will release the previous notes and hold the new ones. The trigger for when to hold/capture and when to release is based upon any note being held. Once the last note is released and a “no note held” condition exists, the system begins watching for new notes to release and re-hold.

Trap Lines

In a theater organ, a single contact on every key is ganged together to form a "trap." This is a wire that is energized any time a single key on the keyboard is energized. Similarly, in Opus-Two, a "trap check" is performed on each keyboard on each pass. If any keys are held down, the trap line is energized. This is useful for a variety of things, such as percussive effects. Timers can even be stoked by trap lines. The following trap lines are active in all chamber builds:

Ac_Trap (Swell)	A2_Trap (Choir)	Gr_Trap	G2_Trap
So_Trap	Pd_Trap	P2_Trap	Bm_Trap

A tap cymbal example, assuming a tab is marked and defined as "Great_Cymbal":

```
stp_ctrl ( Gr_Trap & Great_Cymbal , 4 , 11 )
```

This example will energize card 4, pin 11 every time the trap line is energized (a key held down) and the cymbal tab is engaged.

Pizzicatto

Pizzicatto keying is timed based upon the note being pressed. Two types of pizzicatto keying exist within Opus-Two, pre-pizzicato and post-pizzicato. If a timer exists to play a key for 100 milliseconds after the note has been pressed, the pre-pizzicato will be active from the time the note is pressed until the timer is satisfied. Post-pizzicato will be active from the moment the timer is satisfied until the key is released. An example of pre-pizzicato keying being used on a set of chimes:

```
-- Great Chimes
  stp_seg_pz  ( Ch_Chimes                      , p_8 , ch_pz_k , 4 )
  rank_trim   ( 22 , P_4 + 7 )  -- Offset to make note start right
  rnk_seg     ( 25 , 15 , 25 )
```

The stp_seg_pz line has the added byte to the end which tells the pizzicato timers how long to allow each note to play once it has been keyed.

The rank trim pitch is p_4 + 7. Referring to the custom pitches and couplers (several pages back), it can be seen that p_4 is actually a value of 37, and since (37 + 7 = 44), and on the table a value of 44 is the same as p_2_2f3, the rank could actually be defined as a p_2_2f3, and it would play correctly.

Reiterations

Reiterating ranks (or controls) continue to release and play while being held on at the console. Reiterating ranks release and rekey each note with an independent timer (started when the note is pressed). Reiterating controls (such as a fire gong) will continue to play as long as the control bit is active.

A reiterating control looks like this:

```
--              trigger         speed card pin
   reit_ctrl ( drum & ac_trap , 6 , 10 , 18 )
```

A reiterating rank plays like any other rank with a single line added to it:

```
-- Glock
   Stp_seg ( Gr_Glock             , p_4   , Gr_k )
   Stp_seg ( So_Glock             , p_4   , So_k )
   Rank_trim      ( 49 , P_8  )
-> reit_rnk_cond ( stop_110 , 49 , 60 )
   orgel_rnk_seg ( 61 , 13 , 1  )
```

The added “reit_rnk_cond” line has 3 variables passed to it. The first is the condition to activate the reiteration. If this is off the rank will play as normal. If this is on the reiteration will begin. Second variable (49 in this case) is the number of notes/timers that the reiteration will use. This is typically the same value as the rank trim above it. The third value is the speed adjustment (how long a complete cycle of on-off is before it should repeat).

Console Controlled Pizz/Reit Speeds

At this point, it is understood from the previous pages that this will create a pizzicato effect with a timer value of 4 cycles:

```
-- Great Chimes
  stp_seg_pz ( Ch_Chimes , p_8 , ch_pz_k , 4 )
  rank_trim  ( 22 , P_4 + 7 )  -- Offset to make note start right
  rnk_seg    ( 25 , 15 , 25 )
```

We can also create a reiterating control with a speed of 6 cycles:

```
--             trigger         speed card pin
   reit_ctrl ( drum & ac_trap , 6 , 10 , 18 )
```

Or reiterate an entire rank with a speed of 60 cycles:

```
-- Glock
   Stp_seg ( Gr_Glock            , p_4  , Gr_k )
   Stp_seg ( So_Glock            , p_4  , So_k )
   Rank_trim     ( 49 , P_8  )
-> reit_rnk_cond ( stop_110 , 49 , 60 )
   orgel_rnk_seg ( 61 , 13 , 1  )
```

It may be desirable during initial setup to be able to control these values from the console as opposed to recompiling them repeatedly to find the desired value. There are two common methods to accomplish this, one can be semi-permanent and one is absolutely temporary.

The Very Temporary Way

The absolutely temporary method involves linking the memory level to an expression byte in the console and varying the console memory level until satisfactory results are obtained.

If the console has this line immediately before the first stp_cplr:

```
exprb3 = mem_level
```

And the chamber reiterating control uses exprb3 as the speed like this:

```
-- Glock
   Stp_seg ( Gr_Glock            , p_4  , Gr_k )
   Stp_seg ( So_Glock            , p_4  , So_k )
   Rank_trim     ( 49 , P_8  )
   reit_rnk_cond ( stop_110 , 49 , exprb3 )
   orgel_rnk_seg ( 61 , 13 , 1  )
```

It is then easy to adjust the value being passed to that reiteration procedure by simply adjusting the memory level on the console. This is obviously very temporary, but allows the configuring technician an easy way to change the value without recompiling. Once a satisfactory value is obtained, it can easily be hardcoded into the file and never changed again.

The Semi-Permanent Way

Inside the console controller is a “PzReit Table” of 64 individually settable byte values. These bytes are stored inside the console controller and sent to the chamber every time one of them is changed as well as upon startup. It is possible to use these bytes to control various chamber functions and this makes them (in a sense) settable from the console.

If the console has this line in it’s preamble:

```
use_reits = high
```

And the chamber reiterating control uses any PzRt value (1..64) as the speed like this:

```
-- Glock
   Stp_seg ( Gr_Glock            , p_4  , Gr_k )
   Stp_seg ( So_Glock            , p_4  , So_k )
   Rank_trim     ( 49 , P_8  )
   reit_rnk_cond ( stop_110 , 49 , PzRt1 )
   orgel_rnk_seg ( 61 , 13 , 1  )
```

The special menu in the console to adjust PzRt values will then be usable to adjust the values passed to these parameters.

Floating Divisions

This toolset has 4 floating divisions. Be aware, the buffers for them are using Card Out space (for now), so it is important that if you are using a large number of cards, you understand that sharing of the space. Cards 23-24 is float1, 25-26 is float2, 27-28 is float3, 29-30 is float4. This knowledge is only an "FYI" and doesn't affect how you use the procedures inline. Very few chamber controllers (if any) are driving even 20 cards from a single controller (and using floating divisions), so this is rarely an issue.

An example of using the floating divisions:

```
-- Card 7 Vox Choir Floating Primary 1-73
  chest_enable = ( RG06_Vox_Chorus_on_Choir | RG05_Vox_Chorus_on_Great |
                   RG04_Vox_Chorus_on_Swell | RG03_Vox_Chorus_on_Solo  |
                 RG02_Vox_Chorus_on_Ethereal | RG01_Vox_Chorus_on_Stentor )

  if chest_enable then
    stp_cplr ( high                        , fl1_k , fl1_k , p_0 )
    stp_cplr ( high                        , fl2_k , fl2_k , p_0 )
    stp_cplr ( RG06_Vox_Chorus_on_Choir    , fl1_k , ch_k , p_8 )
    stp_cplr ( RG05_Vox_Chorus_on_Great    , fl1_k , gr_k , p_8 )
    stp_cplr ( RG04_Vox_Chorus_on_Swell    , fl1_k , sw_k , p_8 )
    stp_cplr ( RG03_Vox_Chorus_on_Solo     , fl1_k , so_k , p_8 )
    stp_cplr ( RG02_Vox_Chorus_on_Ethereal , fl1_k , et_k , p_8 )
    stp_cplr ( RG01_Vox_Chorus_on_Stentor  , fl1_k , st_k , p_8 )
    stp_cplr ( RH01_Vox_Choir_to_Vox_Choir_4  , fl2_k , fl1_k , p_4  )
    stp_cplr ( ! RH02_Vox_Choir_Unison_Off    , fl2_k , fl1_k , p_8  )
    stp_cplr ( RH03_Vox_Choir_to_Vox_Choir_16 , fl2_k , fl1_k , p_16 )
  end if
  stp_seg     ( chest_enable              , p_8 , fl2_k  ) -- floating keys
  stp_seg     ( RJ20_Vox_Choir_to_Pedal_8 , p_8 , p2_uk  ) -- non-couple pedal

 rank_trim   ( 73 , p_8 )
 rnk_seg_out ( 73 , 7  , 1  , fmt_c_a )    -- 61 notes
```

In this example, first, the floating stops are checked to see if any are on with chest_enable. If one is on, then fl1_k and fl2_k are cleared, giving an empty working space. Each keyboard is then copied into float1, and fl1_k uses the floating couplers to create coupled keying in fl2_k. A single stp_seg then outputs the result of all the coupling. The vox choir to pedal is listed as a separate stp_seg so that the coupling doesn't affect it.

Re-arranging output pins

A couple of concepts come together to be helpful here:

1) Every single bit of the input card buffers is defined (in1_1 through in30_64).
2) When the first stp_cplr is defined, the entire input card buffer space is cleared.
3) This same exact memory space is used for the output card buffer space.
4) The bit definitions (in1_1 through in30_64) are still there and still pointing to bits in the (now output) buffer.
5) All that being said, a statement like this is possible (despite how illogical it sounds):

```
if in1_2 then in1_15 = high end if
if in1_3 then in1_14 = high end if
if in1_4 then in1_13 = high end if
if in1_5 then in1_12 = high end if
if in1_6 then in1_11 = high end if
if in1_7 then in1_10 = high end if
if in1_8 then in1_9  = high end if
```

This will take the first byte of output card 4 and invert it (upside down) onto the next byte.

Building Inline Code

Bit_Tst, Bit_Set, and Bit_Clr

One of the most flexible parts of Opus-Two is the ability to very quickly determine the state of any bit in any buffer. This ability is accessible within the config file by a simple procedure. This procedure is a true bit check and returns true or false. The third parameter is the zero-based bit number.

```
if bit_tst ( Swell >> 8 , Swell , 12 ) then
  [do something]
end if
```

Much like the ability to quickly test whether a bit is on or off, Opus-Two can turn a bit on.

```
if [some condition] then
  Bit_Set ( Pistons >> 8 , Pistons , 54 )
end if
```

Or off.

```
if [some condition] then
  Bit_Clr ( Pistons >> 8 , Pistons , 54 )
end if
```

Low_Bit_Check and High_Bit_Check

Opus-Two can check a buffer and return the value of the lowest active bit. This procedure returns a byte/numeric value. The procedure will return 255 if no bits are found active.

```
--                              Buffer                  # of bytes to check
   saved_misc = low_bit_check ( Pistons >> 8 , Pistons , 8 )
```

This companion procedure will check for the highest active bit in a buffer. Again, 255 is returned if no bits are found active.

```
--                              Buffer                  # of bytes to check
   saved_misc = high_bit_check ( Pedals >> 8 , Pedals , 4 )
```

Troubleshooting Hints and Tricks

Combination Action Unresponsive

If the combination action is unresponsive, there are a few things to check.

1) Does General Cancel work?
    a) If yes, then enter common memory to check for a stuck piston before proceeding. Fix any stuck pistons.
    b) If no, continue...
2) Is the piston checking in (check from common memory)?
3) Does common memory see the cancel piston?
4) Are the IO cards sending pulses (check lights)?
5) Is combination_action = high in config?

Chamber Doesn't Play

- Verify at the console that stops are being read and show up in the screen and that keying is being read and shows up in the screen.
- It may seem obvious, but it is worth checking with an ohm meter to make sure the data is correctly connected (A to A, B to B, C to C).
- Is console LCD/VFD display working?
    - If yes, that verifies the console is indeed compiled as a console.
    - If no, it may be compiled as a chamber.
- Are the chamber controller LEDs indicating running software (one LED blinking)?
    - If not, reload software. If problem persists, recompile and reload.
- Is the other chamber LED indicating data error (mostly solid LED)?
    - If no, the chamber controller is properly receiving data. Check config.
    - If yes, there is a data integrity problem.
- See if unplugging the three wire data plug changes the appearance of the LED.
    - If it does, there is some kind of error in the data stream.
    - If it doesn't, the data isn't likely getting there in the first place.

"Expected if but found procedure instead" error during compile

- This is typically indicative of an open if statement somewhere in the config file. Because the compiler isn't a mind reader, it doesn't know where the user intended the statement to end, so it will not accurately show where the problem is.

Compiling Completed Files

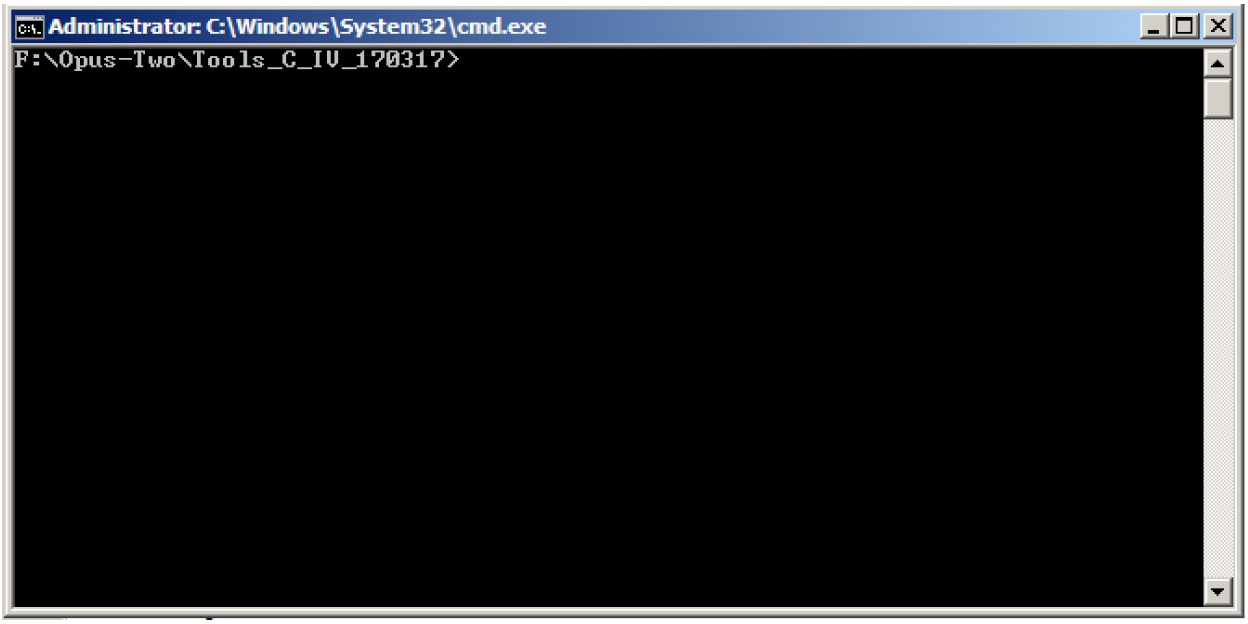
This document assumes the user is writing config files using the included “Jaledit” application. While any text editor (such as Ultraedit) can be used and works fine, these instructions apply only to Jaledit users. Power users are free to write batch scripts to do this work for them, but that exceeds the scope of this document.

Note: The compiler will only compile one file at a time. Do not attempt to have multiple compile sessions running at once. The results will not work as expected.

Getting to the command prompt

In the Tools menu in Jaledit, select “Command Prompt”.

This window will open:



Within Jaledit, if multiple documents are open, it is important to know that Jaledit will load the directory for the active tab when the Tools/Command Prompt is chosen. If multiple documents are open, it is possible to inadvertently open the wrong directory. If an older version of the file exists in the wrong directory, a compile will complete without error and can be very confusing.

Compiling a C-IV console

The following string must be entered for a console compile:

opus_two[space][filename][controller/build][switch]

So, for example, a file called “dd_test_console.jal” would be entered this way:

opus_two dd_test_console c_iv_a

If the compile errors, a hex file will not be created. If the compile completes successfully, a few statistics about the compile will display, and a hex file sharing the same ame as the jal file will appear in the directory with it. This should be loaded using a PicKit2/PicKit3.

Compiling a C-IV chamber

The following string must be entered for a chamber compile:

opus_two[space][filename][controller/build][switch]

So, for example, a file called “dd_test_chamber.jal” would be entered this way:

opus_two dd_test_chamber c_iv

If the compile errors, a hex file will not be created. If the compile completes successfully, a few statistics about the compile will display, and a hex file sharing the same ame as the jal file will appear in the directory with it. This should be loaded using a PicKit2/PicKit3.

Version Change Log

1.6

Thorough editing for grammar and spelling.