This section defines the data files which specify what instruments are available. The precise definition of the file syntax is provided in Backaus-Naur form below. It must

a) begin with a "type" field consisting of `type=` followed by a description of a type which the system knows. This can be a constant, shorthand for one of the default generic types (at the moment: `MIDI`) or a file name (ending in `.ck`). In the case of a file name the system will check the first line for `// type=`. It will then attempt to match the remainder of the line to a constant defined in the `Server.ck` file, at which point it should add the file to the virtual machine and instantiate an object. Note that this code will have to be added to `Server.ck` when a new instrument is added by a `.ck`.

b) follow this with a name, specified by `name=`*name*

c) a MIDI file will now look for a port, specified as `port=`*number or name* (if the name of the port is specified, it can be in quotes, but it is not required)

d) follow this with 0 or more translation lines. The translation lines are of the form `input=message=output` where input is the input midi status byte (disregarding channel), message is the OSC message that it becomes and output is the MIDI message sent to the instrument. The osc message will always have the typetag `ii` with the remaining two input midi bytes (or 0 if it is a smaller message) sent as arguments. In specifying the output message the special terms `$1` and `$2` can be used to refer to the first and second arguments of the osc message. The input status byte (and following =) is optional, omitting it will cause the server to still listen for the specified OSC message and perform the appropriate transform, but will not set up the client to generate the OSC.

Note that messages that do not begin with the name of the instrument will have it prepended. Also if the default messages are not specified, they will be added. A file with 0 translation lines would cause the server to instantiate the object specified, and may therefore be of some use.

0.1 Grammar

⟨*file*⟩ ::= ⟨*type*⟩ ⟨*linebreak*⟩ (⟨*comment*⟩ ⟨*linebreak*⟩)* ⟨*name*⟩ ⟨*linebreak*⟩ (⟨*comment*⟩ ⟨*linebreak*⟩)* (⟨*file-element*⟩ | ⟨*comment*⟩ ⟨*linebreak*⟩) *

⟨*type*⟩ ::= ‘‘`type=`’’⟨*type-string*⟩ [⟨*comment*⟩] ⟨*linebreak*⟩

⟨*type-string*⟩ ::= ‘‘`MIDI`’’

⟨*name*⟩ ::= ‘‘`name=`’’⟨*name-string*⟩ [⟨*comment*⟩] ⟨*linebreak*⟩

⟨*name-string*⟩ ::= ‘`[a-zA-Z][a-zA-Z0-9_]*`’

⟨*file-element*⟩ ::= ⟨*midi-port*⟩
| ⟨*translation*⟩

⟨*midi-port*⟩ ::= ‘`port=`’ ⟨*midi-port-desc*⟩

⟨*midi-port-desc*⟩ ::= '`[a-zA-Z][a-zA-Z0-9]*`'
| '`[0-9]+`'

⟨*translation*⟩ ::= [⟨*input-status-byte*⟩ '=']⟨*osc-message-desc*⟩ '=' ⟨*output-message-desc*⟩
[⟨*comment*⟩] ⟨*linebreak*⟩

⟨*input-status-byte*⟩ ::= ⟨*midi-stat-byte*⟩

⟨*osc-message-desc*⟩ ::= '"'⟨*osc-addr-pat*⟩ ⟨*osc-typetag*⟩ '"'

⟨*osc-addr-pat*⟩ ::= ('/' ⟨*osc-string*⟩)+

⟨*osc-typetag*⟩ ::= ',' ⟨*osc-type*⟩+

⟨*osc-type*⟩ ::= 'i'
| 'f'
| 's'
| 'b'

⟨*osc-string*⟩ ::= '`[^\0]+`'

⟨*output-message-desc*⟩ ::= ⟨*midi-message*⟩
| future message types

⟨*midi-message*⟩ ::= ⟨*midi-stat-byte*⟩ ',' ⟨*midi-data-byte*⟩ ',' ⟨*midi-data-byte*⟩

⟨*midi-stat-byte*⟩ ::= 128–255

⟨*midi-data-byte*⟩ ::= 0–127
| ⟨*osc-arg*⟩

⟨*osc-arg*⟩ ::= '$'[1-2]

⟨*comment*⟩ ::= '#' any characters

0.2 Example

0.2.1 Basic MIDI File

```
type=MIDI                          #this must come first
name=Example                       #and this second
# and now we are free of order
port="IAC Driver 1 Bus 1"          #could be a number
# this would be enough for only the standard interface
128=/noteoff,ii=129,$1,$2          #pass a note off to channel 2
224=/pitchbend,ii=225,$2,$2        #only send one byte
144=/note,ii=145,$2,$1   #redefine default note, change order
/block,f=160,64,$1       #message without MIDI input
```

0.2.2 Example Addition to Class Hierarchy

```
//type=EXTENDED_TYPE
/* ^^ has to be first line
 * EXTENDED_TYPE must be defined in Server.ck
 * and this file must have been added to the VM by
 * Master.ck
 *
 * a MIDI instrument that sends each
 * message on a different channel */
public class ExtendedType extends MidiInstrument
{
    int chan;
    // need to do some init
    fun int init( OscRecv input, FileIO file )
    {
        // file contains this file
        // we will just hardcode the default msgs
        "ext" => name; // name is a field in Instrument
        string patterns[0];
        MidiMsgContainer noteMsg;
        MidiDataByte d1, d2;
        d1.set( MidiDataByte.INT_VAL );
        d2.set( MidiDataByte.INT_VAL );
        noteMsg.set( 144, d1, d2 );
        noteMsg @=> transform_table["/ext/note,ii"];

        MidiMsgContainer cntMsg;
        MidiDataByte d3, d4;
        d3.set( MidiDataByte.INT_VAL );
        d4.set( MidiDataByte.INT_VAL );
        cntMsg.set( 176, d3, d4 );
        cntMsg @=> transform_table["/ext/control,ii"];

        // choose a MIDI port for output
        setMidiPort( 0 );

        // let Instrument set up OSC listeners
        patterns<<"/ext/note,ii";
        patterns<<"/ext/control,ii";
        return __init( input, patterns );
    }

    fun void handleMessage( OscEvent evt, string addrpat )
    {
        if ( transform_table.find( addrpat ) )
        {
            // get message define
            transform_table[addrpat].getMsg( event )
                @=> MidiMsg msg;
            if ( msg != null )
```

```
            {
51              chan += msg.data1;
                16 %= ++chan;
53              mout.send( msg );
            }
55      }
    }
57  }
```