Specification **SP059**

Macotec machines interface

protocols and formats

Created by:	2017-09-19	MG
Checked by:	2024-10-31	MG
Approved by:		
SP059 - Macotec machines interface.odt		
Size	*Pages*	*Revision*
A4	155	16.23 (2025-06-24)

Index

Table of Figures

1 Introduction

1.1 Objectives

This document details the MACOTEC proprietary protocols and formats designed to interact with their machines.

1.2 Context

MACOTEC produces a wide variety of flat glass processing machines that can be divided in two main families: *tables* for monolithic (*Float*) glass and *lines* for laminated (*Strato*) glass.

The machines are deployed with an onboard PC, equipped with custom software.

1.3 Target audience

This document may be disclosed outside MACOTEC.
It is intended for:

- Third-party developers

1.4 Writing conventions

- *Emphasized text*: quoting technical terms
- `Fixed-width font`: variables and registers (ex. `Label123`)
- Critical information
- Useful information

1.5 Revisions and updates

An updated version of this document can be found here.

2 Generalities

2.1 Glossary

2.1.1 Common terms

Terms	Description
Float	A monolithic layer of glass; if referred to a machine, a machine designed to cut *Float* glass
Strato	Laminated glass: a sandwich of two glass layers with a plastic (PVB, PolyVinyl Butyral) between; if referred to a machine, a machine designed to cut *Strato* glass
Project	A work unit, or job order, consisting of a set of *Schemes*, aka cutting patterns, applied to a set of *Sheets*, to obtain a list of finished *Pieces*
Scheme	Depending on context, a cutting plan/pattern applied to a *Sheet*, dividing it in a set of *Pieces*, but also an aggregate associating a *Sheet* with the contained *Pieces*/*Shapes* and the *Job* to obtain them
Job	Depending on context, the operations necessary to process a *Scheme* or *Step* on a particular machine, and the data uploaded/synchronized on the machine describing those operations
Sheet	A rectangle of a certain material characterized by a width, height, thickness and a material name
Repetition	A single *Scheme* can be applied to multiple *Sheets*: in that case we will say that it has repetitions, or multiplicity greater than one
Subsheet	*Part* of a sheet obtained after a cut that contains further nested cuts
Stripe	A *Subsheet* obtained with a vertical cut applied on the entire *Sheet*, aka the *Product* of a first-level X cut
Cross-cut	A vertical cut applied to the entire *Sheet* from edge to edge, obtaining a *Stripe*
Cutting line	An oriented path that applied to a *Sheet* divides it into two parts
Piece	Part of a *Sheet* containing no further cuts and constituting the processing result
Scrap	Part of a *Sheet* containing no further cuts and not part of the desired *Piece* list, so considered a waste; when narrow it is also called *Trim*
Trim	A narrow *Scrap* that runs the entire length of the *Sheet/Subsheet*
Shape	Processing trajectories inside a *Primitive Piece*
Primitive	A rectangular *Piece* associated with a *Shape*
low-E	A thin coating applied to glass to reduce the amount of ultraviolet (UV) and infrared (IR) light that passes through without compromising the amount of visible light
TPF	A Temporary Protective Film that must be removed before low-E deletion
Scoring	Etching a scratch along a *Path* with a carbide wheel pressed on the glass with some force
Opening	Application of a bending moment along a scored line to cause its controlled fracture (breakout), typically performed with a wooden bar rising under the glass.
Optimizer	Something that converts *Sheets* and *Pieces* into *Schemes*, creating *Projects*
HMI	Human-Machine Interface: the *GUI* program that allows the operator to interact with the machine and process *Projects*
GUI	Graphical User Interface: a program that allows users to interact through buttons and other graphic elements
Synchronize	The *HMI* action of aligning the *Job* uploaded/synchronized on the machine with the selected one in the *GUI*
Desynchronize	The *HMI* action to invalidate the current *Job* in the machine
CNC	The intelligent device that controls the machine
Axis	Denotes the drive, motor and mechanical transmission assembly
Rack	Storage stand for collecting and organizing finished *Pieces*
Table	Often used as a synonym to refer to the machine itself, since it consists of a large table on which the glass is placed

Terms	Description
Line	Set of modules along the glass handling flow, usually composed by a *Loader*, transport module, cutting module, breakout module
Loader	System located upstream of the *Line*, responsible for overseeing the loading of *Sheets*
Load	The act of transferring a new *Sheet* on the machine

2.1.2 *Strato* specific terms

Terms	Description
Step	A unit of processing on *Strato* machines, characterized by a cut and possibly the handling operations that precede and follow it
Job	All the data uploaded/synchronized into the machine necessary to work a certain *Step* of a certain *Scheme* for a selected material
Cutting sequence	On *Strato* machines the overall sequence to cut a glass consisting of scoring, opening and detaching
Detaching	Sequence of operations to separate a scored and opened cut on a *Strato* glass. Consists of heating, pulling, and cutting the inner plastic layer
Aligning	Placing the glass to align the cut to the cutting bridge, usually done by pushing the glass with moving blocks
Processing	In a certain *Step* all the operations performed on the glass after *Aligning* and before the final movements, like low-E deletion, *Scoring*, *Opening*, *Detaching*
Product	Cut part remaining on the alignment side of a *Strato* machine
Remnant	Cut part remaining on the detachment side of a *Strato* machine
Feed	Module that acts as a stack for the initial *Sheet* and the *Remnants* to be processed
Buffer	Additional area to decouple the operations in the *Processing zone* from the stacked *Subsheets* in the *Feed*
Detach	Area immediately upstream of the cutting bridge, where a traction mechanism pulls the glass to separate the cut. After a cut the *Part* that remains on the *Detach* side is called *Remnant*
Cutting bridge	The overall mechanics, developed vertically, that allows the movement of the tool carriages
Cut area	A void area, where the glass can fall, through which pass the processing tools that
Cut axis	The path traced by the scoring heads, coincides with the y-axis
Align	Zone where moving blocks align a cut to the *cut axis*. After a cut the *Part* that remains on the *Align* side is called *Product*
Processing zone	The union of *Buffer*, *Detach*, *Cut area* and *Align* zones, where occurs the processing of the current *Step*
Out-zone	Area at the end of the *Line* where finished *Pieces* are retrieved by the operator, located downstream of the *Align* module. Sometimes referred as *Line-end*.
PVB Blade	Tool to cut the plastic interlayer (PolyVinyl Butyral) inside laminated glass
Head deletion	A Step consisting on just a low-E wheel pass on a *SubSheet* edge

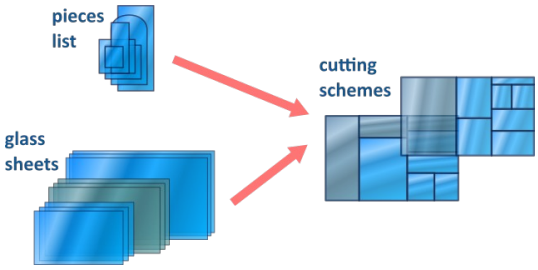
2.1.3 *Float* specific terms

Terms	Description
Job	All the data uploaded/synchronized into the machine necessary to work a certain *Scheme* for a selected material
Processing	All subsequent operations after the start of a *Scheme*
Path	A working trajectory on the glass, composed by a sequence of arcs and lines, that can be associated with a tool
Stage	A sequence of *Paths* involving the same tool
Rectilinear	Straight cuts parallel to one of the *Sheet* sides, thus aligned with reference axes
Curvilinear	Any cut not *rectilinear*

Terms	Description
Unidirectional cuts	Cuts processed imposing a certain direction on the tools (for example toward the positive x/y)
Squaring	In *Float* machines the operation after loading a *Sheet* on the table to ensure the *Sheet* is perfectly horizontal, aligned with the machine XY axes
Structural deletion	Low-E deletion performed on an area wider than the low-E wheel, involving multiple passes of the tool

2.2 Some terms

Glaziers maintain an inventory of glass *Sheets* in various sizes and materials; when a customer requests specific *Pieces*, an optimization process determines the most efficient cutting plan to extract the required *Pieces* from the available *Sheets*.



A single cutting plan, associated with a *Sheet*, is called *Scheme*; a certain *Scheme* can be repeated on several *Sheets*: in this case the *Scheme* has a *multiplicity* greater than one.

A *Project* consists in a list of *Schemes* produced with a single optimization process; in general it can involve several *Sheets* of various formats and materials, to obtain *Pieces* for different customers.

The terms *Scheme* and *Sheet* cannot be used indifferently, a certain *Scheme* can be applied to more *Sheets*.

A *Sheet* is a rectangle of a certain size and material.

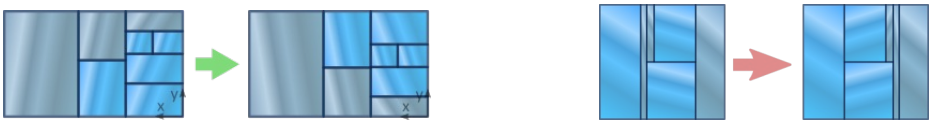
A *Scheme* is a set of cuts applied to a *Sheet* that divides it into smaller *Pieces*.

The main cuts are *rectilinear*, straight lines parallel to the sides of the *Sheet* that cross the glass from side to side, dividing it into smaller rectangles; once all the cuts have been applied, the resulting rectangles are either finished *Pieces* or *Scraps*.

The desired piece could be a *Shape*; in this case its axis-aligned bounding box, called *Primitive* rectangle, is a *Piece* obtained as described, that then will be further processed to obtain the *Shape*.

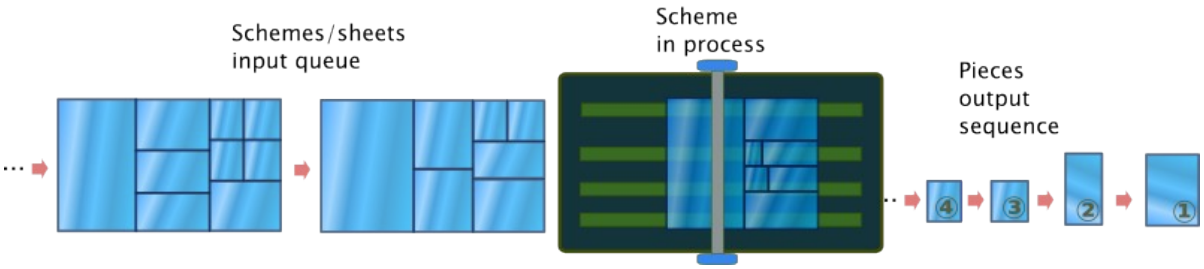
The *Schemes* are generally created automatically by an optimization algorithm, whose main criteria are to minimize waste (*Scraps* area) and ensure workability; the latter criterion depends on the characteristics of the machine and is normally in conflict with the first, and acquires particular significance in the case of fully automatic machines to minimize the manual intervention of the operator.

The *Schemes* created by a generic optimization algorithm may then tuned to meet particular machine requirements; this usually involve just small local changes that have minimal impact on the overall pattern. This tuning may have to adhere to certain constraints, such as maintaining the expected output sequence of *Pieces*.



From each *Scheme* the *HMI* generates a *Job* in order to process it on a particular machine. Often, we will loosely use the term *Scheme* to refer to an aggregate that includes the cutting plan, the *Sheet* it is applied to, the list of desired *Pieces*, and the *Job* for processing it on a particular machine.

Once a *Project* is transmitted to the machine, the machine can process it *Scheme* by *Scheme*:



2.3 Cuts tree/*Scheme* basics

In glass cutting, a cut must start and stop on the *Sheet* edges or in correspondence of another (perpendicular) cut: this is called *guillotine* cut and is a technological requirement in order to open it.

This implies that each cut divides the glass in two *Parts*, and that's the reason why *guillotine* cuts are naturally represented by a binary tree, and *Schemes* are sometimes referred as *cut trees*.

Note that binary tree is inherently recursive by nature.

Each cut is a node of the tree, and can have siblings and children cuts. A cut without children is a *leaf*, and obtains just *Pieces* or *Scraps*.

Each cut divides the *Subsheet* to which is applied into two *Parts*. The *Part* enclosed by the cut dimension is called *Product*, while the other is the *Remnant*.

The dimension associated with a cut is the *Product* width.

Each cut applies to the *Remnant* of its preceding sibling, if there it one, otherwise it applies to the *Product* of its parent, or, if there's no parent, to the initial *Sheet*.

2.4 XYZ notation

In glass cutting industry the typical notation to represent such trees is associating a letter, representing the cut level, with the dimension of the cut. The letters generally adopted are:

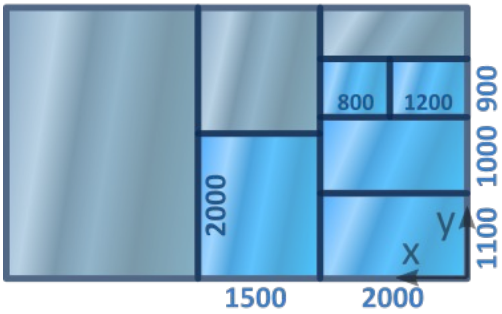
Level	1	2	3	4	5	6	7	8
Letter	X	Y	Z	W	V	A	B	C

The *Product* of an X (level 1) cut is called *Stripe*. The *Remnant* of the last X cut is the main *Scrap* of the *Sheet*.

It's rare to have *Schemes* with a cut nesting level greater than 4.

As example consider the following *Scheme*:

The adopted left-handed reference axes is not important, the same *Scheme* may be drawn flipped horizontally.



X 2000
Y 1100
Y 1000
Y 900
Z 1200
Z 800
X 1500
Y 2000

Note that all the lengths are relative and the *Sheet* size, to which the cuts tree is applied, is not specified.

The ordering matters: considering the X cuts, the first is 2000 and the second is 1500.

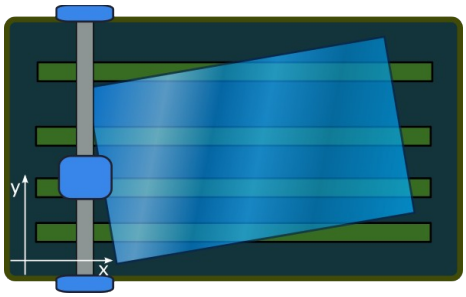
The *Product* of the first cut is a *Subsheet* whose width is 2000, which contains three children cuts. The first two children are *leafs*, while the last has two children, the second cut applies to the *Remnant* of his preceding sibling, and so on.

2.5 *Float* machines

A machine designed to process monolithic (*Float*) glass is informally called *Float* machine.

The *Float* glass is cut scratching its surface with a very hard carbide wheel (scoring) and then applying a bending moment to open the cut (open/breakout). A cut trajectory can be also *curvilinear*; in this case additional cuts are usually added to aid the breakout.

2.5.1 Structure of *Float* machines



This family of machines consists in a carriage equipped with various tools for glass processing (scoring heads, low-E wheel, vinyl knife, *TPF* scraper, marker, labels applicator, glass probe, …). The carriage can move in a horizontal plane above a table where lays the glass sheet. The movement is composed by the two orthogonal axes X Y, mechanically realized with a *Bridge* and a *Carriage*.

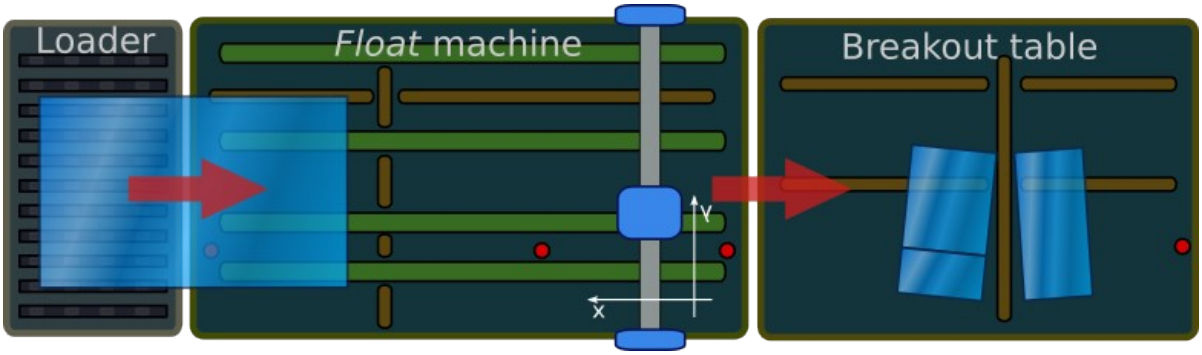
A third axis (usually referred with Z) represents the tool orientation, usually tangent to the XY trajectory.

The tools push on the glass with a force controlled by pneumatic actuators.

Depending on the machine type, the glass sheet can be automatically shifted using conveyors mounted on the table.

A *Job* consists in a file containing the various tools trajectories, accordingly to the *Scheme* layout.

A *Float Line* is usually composed by a *Loader* upstream that provide the *Sheets* to the machine; after scoring them, the *Sheet* is then moved out to a downstream breakout table, where the operator opens the cuts and retrieves the finishes *Pieces*.

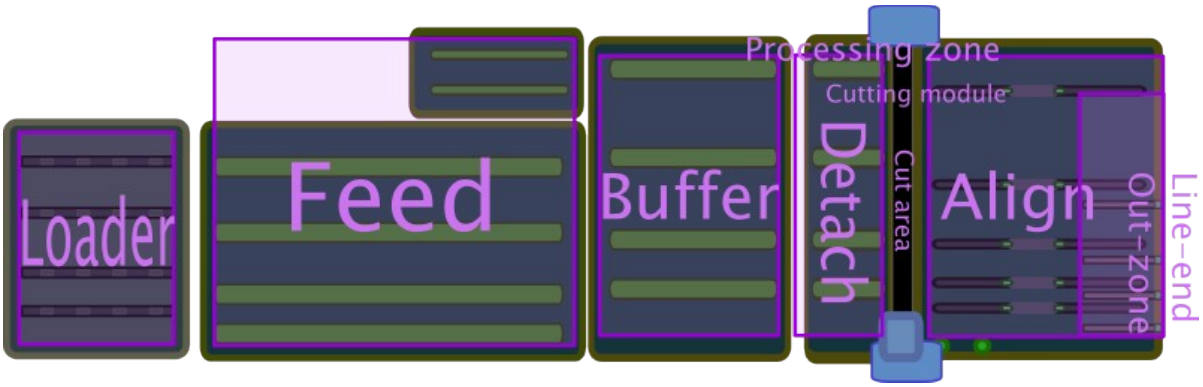


2.6 *Strato* machines

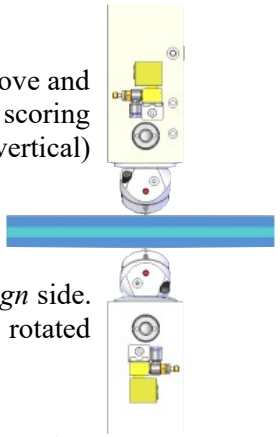
A laminated (*Strato*) glass is a sandwich composed by two glass layers and an inner plastic film (*PVB*); the cut sequence is a little more complicated and involves multiple tools.

2.6.1 Structure of *Strato* machines

Strato machines are generally composed by a *Line* of modules, in the figure below the glass flows from the left to the right.



2.6.2 The cutting module

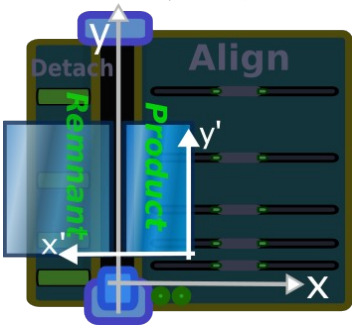


This is the main component of the machine, consists in two carriages, one above and another below the table level (upper and lower carriages), equipped with scoring heads and breakout wheels. The carriages shift on a fixed transversal (vertical) direction Y (the *cutting bridge*).

The *cutting bridge* is equipped also with a heating lamp and a breakout bar to carry out the proper cut sequence.

The cut is aligned pushing the *Sheet* with a series of blocks placed in the *Align* side. Depending on the machine type, the *Sheet* can be automatically placed and rotated using proper devices.

2.6.3 Cutting a laminated glass



Since the cutting bridge is fixed, the glass must be placed to be properly aligned with the fixed *cutting axis* Y.

The left side is called *Detach* side, and the right side is called *Align* side.

After the cut the *Part* that remains on the *Align* side is called *Product*, while the other is called *Remnant*.

2.6.4 *Steps* sequence

To be processed on a *Strato* machine, a *Scheme* must be broken down into a sequence of *Steps*.

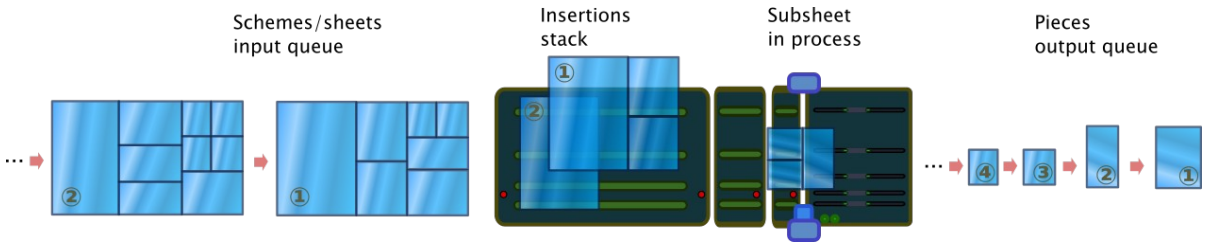
Each *Step* involves positioning the glass for cutting (shifting, rotating, aligning), performing the cut (processing), and executing post-cut movements.

Given a *Scheme*, there are multiple possible processing sequences. Among these, one is particularly well-suited for execution on a certain machine. Therefore, the sequence is crafted based on the specific characteristics of the machine.

A *Job* consists on a fixed sequence of *Steps*, each describing the operations to do on the incoming *Subsheet* (alignment, rotation, score, …) in order to process a *Scheme*.

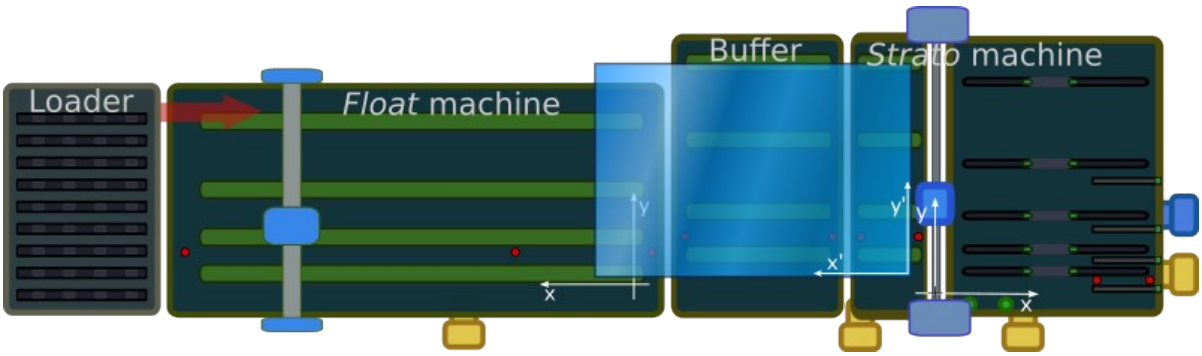
Strato machines work sequentially, delivering finished *Pieces* step by step. The operator must retrieve the *Pieces* and put them in their respective *Rack*, which will be transferred either to the next processing stage or for shipment.

Sometimes is crucial to respect the output sequence of the delivered *Pieces* devised in the original optimization, in that case the *HMI* can still apply local changes to the *Scheme* if they respect those constraints.



2.7 Combined *Lines*

Strato and *Float* machines can be combined in the same *Line*, the *Float* machine acting as the *Feed* table of the *Strato*:



The two machines can interact with each other and collaborate to process a *Scheme*.

Since the two machines can operate in parallel, it makes sense to have separate *HMI* windows for each; both the *HMI* may interact with a supervisor that assigns sub-*Jobs* and monitors when they're completed.

Combined processing refers to dividing and managing a *Job* created for a specific machine (*main machine*) so that it can be partially processed by other machines (*combined machines*).

Depending on the user preferences, synchronizing a *Scheme* on the *main machine* may result in assigning one or more sub-*Jobs* to the *combined machines*. These sub-*Jobs* are referred as pre-processing or post-processing, depending on whether they occur before or after the glass scoring.

Combined processing can be categorized as:

- *Float Projects*
  - *(pre-processed on Strato)*
    Label *Pieces* on *Strato* (printer bridge during *Sheet* load) and then score on *Float*
  - *(post-processed on Strato)*
    Score on *Float* and then perform automatic *Stripes* breakout on *Strato*
- *Strato Projects*
  - *(pre-processed on Float)*
    Low-E deletion/Label *Pieces* on *Float* and then cut on *Strato*
  - *(post-processed on Float)*
    -

Each of these *combined processing* strategies are covered in sect. 9 "Combined processing" below on p.137.

2.7.1 Coordinating the combined workflow

As outlined above, the general idea of a combined work is splitting the original *Job* into a sequence of sub-*Jobs* processed by multiple machines.

This workflow may require a *supervisor* (*Line manager*) that generates the sub-*Jobs* and coordinates the sequence of who does what.

Since each machine only knows how to process a *Scheme* and lacks the concept of combined work (otherwise would violate the principle of separation of concerns), this role must be managed at a higher level.

A possible structure is this:

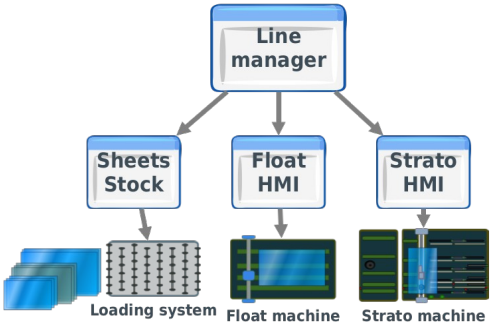


Figure 1: Line manager (combined line)

Where the *Line manager* is an actor aware of all the machines composing the *Line*, splits the original *Job* into sub-*Jobs*, sends them to the relative *HMI*, requests to start the processing of a sub-*Job* and monitors its completion.

As for how splitting, creating and coordinating these sub-*Jobs*, the implementation has maximum freedom, as long as it ensure easy resumption of the combined work by *HMI* navigation, with as much as granularity as possible.

3 Communication protocol

3.1 Context

Originally, the interaction with MACOTEC machines involved a direct connection to the particular controller device, exposing internal details and thus creating a tight coupling between the *HMI* and the machine implementation details. Since the implementations tend to increase over time, this approach has revealed all its downsides in term of scalability and maintainability.

The interaction with next-generation machines is no longer through direct connection to the particular *CNC*, but through a unified interface that abstracts the services of the generic machine while hiding its internal details. This allows past and future machines to be supported with a single communication library, thus without any change on the *HMI* part.

This prompted the creation of *CNC*-agnostic standards to handle and describe our workflows: an extensible common communication protocol, a unified resource dictionary, and a generic descriptor for the machine *Job*.

3.2 Core foundations

Communication takes place over Tcp/Ip *sockets*.

The messages exchanged are `UTF-8`[1] strings that contain write and read operations of fields that represents machine resources. A dictionary of the available fields is maintained in the file `Interface.xml`.

- The string representing the single message must be transmitted in a single `send` to the *socket*. In the typical case, where the *sockets* are local (*buffers* reside on the same PC), there is no particular limit on the length of the single message; *clients* are free to split particularly long messages into several smaller messages to meet any constraints on their *buffer*
- Read operations are merged and performed after write operations; within these two macro-operations the order of the fields is not important and there is no guarantee of the actual sequence of operations on individual fields; reply messages are not guaranteed to keep the ordering of the fields in the request
- The set of operations enclosed in a message is not atomic, it does not constitute a transaction: in case one of the operations fails with a critical error, some operations may have been executed and some other not; *clients* can divide operations into several messages if they need guarantees on single results and sequence
- If in a message there is both a write and a read of the same field, the value read is *implementation-defined*, so best to avoid that

1 Names and values are actually restricted in the `ASCII` subset.

3.3 Syntax

A message is generally divided into two parts: a *header* and possibly a *body*.
Those parts are delimited by the following control characters:

- [␁] (*start of heading*, `0×1`) indicates the start of the message
- [␂] (*start of text*, `0×2`) indicates the end of the *header*/start of the *body*
- [␃] (*end of text*, `0×3`) indicates the end of the message

[␁]<*header*>[␂]<*body*>[␃]

It is allowed to have an empty body: in that case the control character [␂] can be omitted:

[␁]<*header*>[␃]

Both the *header* and *body* consist of a sequence of fields[2] `name=value` (or just `name` in case of unspecified value) separated by the character '`;`' (*semicolon*, `U+003B`).
If a value is specified, it is separated from the respective name (unique identifier or key) with the character '`=`' (*equal sign*, `U+003D`):

```
[␁]name1=value1;name2=value2[␂]name3=;name4=value4;name5[␃]
```

- Empty values such as `name3=;` are allowed; Note that empty value (`name3=`) and unspecified value (`name3`) are not equivalent, having different semantics
- Spaces between the separators are allowed: `name=val; name = val ;`
- The ordering of fields is not meaningful, they may be reordered in replies; in general the sorting is *implementation-defined* and can change from one version to another
- For the full dictionary of available fields always refer to `Interface.xml` file, deployed with the server executable

2 Hereafter we may also refer to them by the terms *resources* or *keys*

3.3.1 Field names (keys)

Field names are unique, case-sensitive, are never enclosed in double quotes and cannot contain spaces or special characters used by the protocol (like `, ; =`).

Names beginning with a non-alphabetic character (`$ # & @ %`) represent special resources.

3.3.2 Field values

The value literals can be numbers, strings or their sequences:

- Strings: although not mandatory, are generally enclosed by '""' (*double quotes*, `U+0022`) to allow spaces and protocol characters. It's not possible to escape double quotes within the literal[3].
  - `name="string literal"`
- Booleans: *true* and *false* are expressed with the literals `0` and `1`
  - `name=1`
- Integers: the default base is `10`, hexadecimal base with prefix '`0x`' is supported
  - `name=42`
  - `name=0x1F`
- Floating point: decimal separator must be '`.`' (*dot*, `U+002E`), no thousands separators allowed; exponential notation is supported (ex. `1.2E-6`)
  - `name=12.25`
  - `name=-5.342E3`
- Tuple: a pair of values separated by '`,`' (*comma*, `U+002C`) and enclosed in round brackets '`()`' (*left/right parenthesis*, `U+0028/U+0029`)
  - `name=(100,200.3)`
- List: a sequence of values separated by '`,`' (*comma*, `U+002C`). Lists of string literals can be represented by a single string containing the list.
  - `name=1,4,5,6`
  - `name="one,two,three"` preferred to `name="one","two","three"`

Empty values

Non-numeric values (string, list) can have empty values: an empty value can be expressed by having the '`=`' character followed by a separator or control character, or by an empty string `""`. It is also recommended to support the special literal `(null)`:

- `name=[;|␜|␝]`
- `name=""`
- `name=(null)`

> ⚠ Do not confuse an empty value with an unspecified value. `name=` and `name` are not semantically equivalent: in the former is specified an empty value for `name`, in the latter the key `name` is declared without any value.

3 No *escaping* character is provided such as '\' (*backslash*, `U+005C`)

3.3.3 Header

The header is the first part of the message: it starts with the character [SOH] and ends with [STX] (or [ETX] if the body is empty).
The fields allowed in the header are:

<table>
<tr><th>Label</th><th>Type</th><th>Description</th><th>Notes</th></tr>
<tr><td><s>len</s></td><td><s>unsigned int</s></td><td><s>Total length in bytes of the message, i.e., the number of characters including [SOH] and [ETX]</s></td><td></td></tr>
<tr><td>id</td><td>unsigned int</td><td>A non-zero unique identifier of the message, used to associate a reply; once the exchange is consumed it can be reused; it can be reset beyond a certain limit ex. <code>id=1+id%9999</code> (it is not necessary to reach the representable limit)</td><td>Mandatory in requests</td></tr>
<tr><td>rep-to</td><td>unsigned int</td><td>Identifier of the message/request being responded to</td><td>Replies only, allows to associate it to the corresponding request</td></tr>
<tr><td>ret</td><td>int</td><td>Return code/reply error
<table>
<tr><th>Val</th><th>Description</th></tr>
<tr><td>0 RET_OK</td><td>All ok, no errors</td></tr>
<tr><td>1 RET_SYNTAXERR</td><td>Message syntactically invalid</td></tr>
<tr><td>2 RET_RES_UNKNOWN</td><td>Unrecognized resource</td></tr>
<tr><td>3 RET_VAL_INVALID</td><td>Incorrect value</td></tr>
<tr><td>4 RET_WRONG</td><td>Semantically incorrect request (non-compliant or incomplete)</td></tr>
<tr><td>5 RET_RES_UNAVAIL</td><td>Resource not accessible (machine not connected?)</td></tr>
<tr><td>6 RET_VAL_UNAVAIL</td><td>Value not available</td></tr>
<tr><td>7 RET_DENIED</td><td>Access denied (permissions, read/write rights)</td></tr>
<tr><td>8 RET_FAILED</td><td>Execution error (machine not ready, …)</td></tr>
</table></td><td>Replies only, can be omitted if null (<code>ret=0</code>)</td></tr>
<tr><td>msg</td><td>string</td><td>Message string, can indicate a precise command or intent of a request:
<table>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>ping</td><td>Connection test</td></tr>
<tr><td>greet</td><td>Initial message with machine data</td></tr>
<tr><td>connect</td><td>Connection request</td></tr>
<tr><td>disconnect</td><td>Disconnection notification</td></tr>
<tr><td>ready</td><td>Resume work notification</td></tr>
<tr><td>news</td><td>Status change notification</td></tr>
<tr><td>event</td><td>Work event notification</td></tr>
<tr><td>data</td><td>Data/measure notification</td></tr>
<tr><td>done</td><td>Command executed notification</td></tr>
<tr><td><other></td><td>Diagnostic string</td></tr>
</table></td><td>Used for protocol expansions, it is generally omitted in ordinary requests and responses.
It can be used by responses to send a diagnostic message to the requester.</td></tr>
</table>

3.3.4 Body

The body is the part of the message delimited by the characters [STX] and [ETX]; it contains read/write requests for the fields available in the generic machine interface; for the dictionary of available names, refer to the `Interface.xml` file, in which are defined also the data type and expected range; the external program should create a map of these fields by associating their identifiers with its own internal descriptors.

Names beginning with the special character '`$`' (*dollar*, `U+0024`) represent a group of fields (see sect. 3.8 “Groups of fields” below on p.27).

3.4 Requests, replies and notifications

Messages are divided into three types: *requests* (messages that require a reply), *replies* (result of a request), and *notifications* (one-sided messages that do not require a reply).

Request messages must declare a unique identifier (`id` field in the header), consisting of a progressive positive integer; each request awaits a response referring to its identifier.

Replies must declare the identifier of the request to which they are responding (`rep-to` field in the header); they are therefore recognizable by the presence of this field; it is allowed to specify an identifier (`id`) if the reply needs itself a reply. A reply may contain a return code `ret`.

Notifications are message sent without a specific request triggering them, and don’t expect a reply.

In brief:

- Requests: have `id`, have no `rep-to`, always require a reply
- Replies: have `rep-to`, can have `ret` (if `0` can be omitted)
- Notifications: have neither `id` nor `rep-to`, no need to respond

Example – request and reply

⇨ [SOH]id=19[STX]spdovd=180.2[ETX]

⇦ [SOH]rep-to=19;ret=3;msg="cannot set spdovd to 180.2"[ETX]

Example – notification

⇦ [SOH]msg="news"[STX]scheme=1;step=4[ETX]

3.5 Conversation

We refer by the term *client* to the sender of a request and by *server* who receives it and replies. These roles are not fixed: although the *HMI* usually acts as a *client*, it must be ready to handle incoming requests or notifications.

Communication is bidirectional, and both *nodes* may send a message at the same time; in such cases each receiving *node* sends a reply for the received request and continues to wait for a reply to its own sent request.

Every request (message with an `id`) must always be answered with a positive or negative reply.

> The message `id` is unique within the individual *node*: there is no ambiguity if both *nodes* send a message with the same `id`.

3.5.1 Error handling

The *client* must not send a further request until it receives a reply from the previous one; if it does not receive a reply within a certain timeout (default `3 s`), it makes a new attempt until the maximum number of attempts is reached (default `2`). If all attempts fail it closes the socket and sets a brand new connection.

In case of an error reply (`ret` has non-zero value), the requested operations are to be considered as not executed; depending on the error type (see table above) the *client* will proceed to notify the error or retry.

> When digesting an incoming message is advisable to be tolerant: the presence of unrecognized fields should not generate an error, they should be ignored until they will be eventually supported in later versions.

3.6 Connection and authentication

More than one machine may be present: to connect to one of them you must first authenticate with a *connection manager*, that will then assign the proper channel to connect with the desired machine.
The procedure for connecting to a machine:

1) Open a *socket* at the *connection manager* address (value set in the configuration, default: `localhost:23200`)

2) Upon connection the *connection manager* will send a greet message containing the list of available machines and an authentication public key (`char`):
`[␑]id=1;msg="greet"[␒]machines="ActiveW,StarCut";auth-key=0x31[␗]`
(if it does not arrive, close the socket and raise a connection error)

3) Reply with a connection request, specifying your identity (process name), the machine you wish to connect to, the level of authorization you require (`0`:*guest* `1`:*user* `2`:*root*) and a public key[4]:
`[␑]id=1;rep-to=1;msg="connect"[␒]sender="my_hmi.exe";`
`machine="ActiveW";auth-lvl=1;auth-key=0x1A[␗]`

4) If your connection request is accepted, a positive reply will contain the port (the IP address doesn't change) where the communication channel with the requested machine is available:
`[␑]rep-to=1[␒]port=25463[␗]`
Or in case of an error the connection manager may reply with:
`[␑]rep-to=1;ret=4;msg="not authorized"[␗]`
`[␑]rep-to=1;ret=3;msg="unknown machine ActiveW"[␗]`

5) If the request was successful you can close the first socket used to authenticate and open a new one at the communicated port. On this new channel the frames are encrypted[4] with the private key, the examples here are reported already decrypted. Upon connection the machine remains silent, you have to make yourself known by sending a message similar to the following, declaring the name of your process:
`[␑]id=1[␒]sender="my_hmi.exe"[␗]`

6) The machine, if recognizes you, will reply with a greet message containing its characteristics:
`[␑]rep-to=1;msg="greet"[␒]machine="ActiveW";version=4;unit-length=0.001;…[␗]`
The incoming data should be used to self-configure; see next section for details.

7) At this point the connection sequence is finished and you can talk to the machine; the first request will probably be subscribing to status changes:
`[␑]id=2[␒]$subscribed="$status"[␗]`

4 For the algorithms to calculate the public key and messages encryption/decryption, ask directly to MACOTEC

3.6.1 Greet message

On first connection the machine sends a message containing its characteristic data; the meaning of the various fields is described in the `Interface.xml` file.

The order with which the fields are listed is not guaranteed but the content is, and may become richer with new versions.

A list of fields common to all machines types:

`machine`	Machine name
`mach-type`	Machine type (`float`, `strato`)
`version`	Machine *software* version
`mach-capabilities`	Prerogatives and capabilities of the machine, available tools
`using-internal-materials-db`	Declares if the materials processing parameters are managed internally
`unit-length`	Unit used to measure space, expressed in meters usually: `0.001` (millimeters)
`unit-time`	Unit used to measure time expressed in seconds usually: `1` (seconds)
`unit-speed`	Unit used to measure speeds, expressed in meters per second usually: `0.017` (`1/60`, meters per minute)
`unit-force`	Unit used to measure forces expressed in newtons (ex. `3.14` means tenths of a bar, referring to a cylinder `⌀20 mm`) usually: `1` (newtons)

If the units of measurement adopted differ from those declared by the machine, this is the most appropriate time to set them with a specific request:

⇨`[§]id=3;msg="using inches"[§]unit-length=0.0254[§]`

> ⚠ This facility is introduced for future use but not yet supported.

The greet message will contain also specific data depending on the machine type, for further details refer to sections:

7.6 “Strato machine characteristics” below on p.75

8.7 “Float machine characteristics” below on p.107

3.7 Reading and writing fields

An example where is requested to write (*set*) a field `foo` and at the same time to read (*get*) two other fields `bar` and `baz`, followed by two possible replies:

⇨ `[#]id=23[$]foo=15;bar;baz[%]`
⇦ `[#]rep-to=23;ret=0[$]bar=153.3;baz="ciao"[%]`
⇦ `[#]rep-to=23;ret=2;msg="unknown resource baz"[%]`

The ordering of the fields within the message is not meaningful, and the actual sequence of operations is *implementation-defined*, in other words is not guaranteed and can change.

⚠ Best to avoid reading and writing the same field in the same message: the sequence of the two operations is implementation-defined.

ℹ If at least one of the operations requested generates an error (for example it is not possible to read the value of the unknown field `baz`), the execution of the other actions is not guaranteed: some might have been executed and others aborted, in the example above `foo` might have been written, or not.

Requests are bidirectional: for example, the machine might request data from the *client*, in the following example the *Scheme* index and the material currently selected:

⇦ `[#]id=23[$]scheme,glass-id[%]`
⇨ `[#]rep-to=23[$]scheme=3; sheet=2; glass-id="44.1"[%]`

⚠ Replies may reorder the original fields, and include fields not explicitly requested: in this example the responder decided to supplement the data with the value of a `sheet` field not present in the request. The requester acknowledges the presence of the additional information and may use it.

ℹ Note that the reply in this example doesn't contain the return value `ret=0`, which is optional since indicates a positive outcome.

Example reading some data

⇨ `[#]id=24[$]scheme;step;op-status[%]`
⇦ `[#]rep-to=24[$]scheme=2;step=16;op-status="detaching"[%]`

ℹ These are only educational examples: for how to monitor machine status see sect. 3.10 "Monitoring the machine status" below on p.28.

Example reading the emergency list and active messages

⇨ `[#]id=26[$]emg-list,msg-list[%]`
⇦ `[#]rep-to=26[$]emg-list=30800015;msg-list=5,15,234[%]`

Note that the machine passes a sequence of integers: the string and alarm image must be resolved using an alarm description file `Alarms.xml` dependent on the machine type.

Example writing of some fields

⇨ `[#]id=27[$]spdovd=82; cut-recipe="max-pull,no-heat"[%]`
⇦ `[#]rep-to=27[%]`

3.8 Groups of fields

The prefix '$' (*dollar*, U+0024) indicates a group of fields.
A group of fields can be defined in this way:

⇨[§]id=28[ǂ]$my-work-status="scheme,step,op-status"[ǁ]
⇦[§]rep-to=28[ǁ]

Existing groups can be included in the definition list, for example:
$my-status="$status,job-descr"
Defines a group containing all the fields of $status plus the additional field job-descr

Groups are global (except private ones such as $subscribed), they are shared by all clients on the machine: it is easy to detect any conflicts due to the fact that you cannot directly edit a group without first deleting it.

Once defined, it's possible to read all the fields contained in the group:

⇨[§]id=29[ǂ]$my-work-status[ǁ]
⇦[§]rep-to=29[ǂ]op-status="rotating";scheme=2;step=16[ǁ]

A defined group cannot be edited directly, but it can be deleted and redefined.
To delete a group, simply assign it an empty string.

⇨[§]id=30[ǂ]$my-work-status=""[ǁ]
⇦[§]rep-to=30[ǁ]
⇨[§]id=31[ǂ]$my-work-status="scheme,step,op-status,steps-count"[ǁ]
⇦[§]rep-to=31[ǁ]

You can request the list of currently defined global groups with:

⇨[§]id=32[ǂ]$$groups[ǁ]
⇦[§]rep-to=32[ǂ]$$groups="$material,$my-work-status,$status,$units"[ǁ]

There are predefined global groups that cannot be deleted (and thus redefined): $status (machine status), $material (material processing parameters), $units (adopted units of measure).
In addition to global groups, there are special private groups associated with the *client* that have specific effects, such as $subscribed (see sect. 3.10.3 "Subscribing to status change notifications" below on p.29).

3.9 Notifications and requests from the machine

The *HMI* must consider the possibility of unsolicited messages incoming from the machine, and be ready to reply to any request.
Ex: a possible request for the next *Scheme* to be cut:

⇦[§]id=634[ǂ]scheme;sheet;glass-id[ǁ]

In this case the client can respond synchronizing the new data and replying with the new synchronized values.

Ex: a possible notification of a change in the position of the glass on the table:

⇦[§]msg="news"[ǂ]sheet-position=(10.5,23.34);sheet-angle=5.32[ǁ]

Since it's just a notification, no need to reply: the client is free to use or ignore the transmitted information.

3.10 Monitoring the machine status

Tracking the machine current status can be handled in two modes, which can coexist: cyclically sending read requests of the `$status` group (*polling*), and subscribing to status change notifications.

> There is an inherent limit on the "freshness" of the status information, determined by the frequency at which the *CNC* registers are read (a typical period might be `300 ms`).

Both modes refer to the default group `$status`, which contains the fields that characterize the state of the machine.

3.10.1 The `$status` group

This group contains all the fields that characterize the status of the machine (ex. alarms, current operation, work selectors, …). Those fields depend on the machine type and version: any unrecognized field should be tolerated (connection with new versions), and a fallback alternative should be adopted in case of absent fields (connection with old versions). Some fields may or may not be present depending on the status (ex. `scheme-progress` is only present during machining).

> The behavior of some monitored fields depends on the machine implementation. To avoid unnecessary dependencies, do not assume uniform behavior across all machines. Instead, follow the directions in this document precisely to correctly interpret the status.

> For the meaning and interpretation of the read fields, refer to the `Interface.xml` file.

3.10.2 Polling the status

The *client* can regularly send a request to read the predefined group `$status`, which contains all the fields of interest:

⇨ `[§]id=244[§]$status[§]`

⇦ `[§]rep-to=244[§]…;mach-status="ready";…;working=0[§]`

Choose the polling frequency to avoid overburdening the communication channel.

> Due to the inherent limitation of the freshness of the status, it is unnecessary to read it too frequently.

> If subscribed on status change notifications, the polling period can rise significantly to as much as several seconds.

This specific request (reading only the `$status` group) is recognized by the machine as a *polling* request and is handled in an optimized manner.

In case of error response, for example:

⇦ `[§]rep-to=244;ret=5;msg="Machine not connected!"[§]`

It is strongly advisable to remove any (now unreliable) status information from the graphic and wait for the data to return.

In case you need to monitor other fields, although it is possible to send a request such as:

⇨ `[§]id=245[§]$status;added1;added2[§]`

It is not recommended because the *server* would no longer recognize it as a *polling* request and thus it would be handled less efficiently: better request MACOTEC to add the desired field names to the `$status` group.

3.10.3 Subscribing to status change notifications

Clients can request the machine to notify them if there's a change of the value of certain fields, typically the ones grouped in `$status`.

This mechanism minimizes the data traffic and processing load of the client, which has to interpret shorter and less frequent messages, while maximizing the freshness of the state information: it is therefore highly recommended.

The subscription is triggered by defining the special group `$subscribed`, typically assigning the `$status` group:

⇨ `[§]id=2[§]$subscribed="$status"[£]`

⇦ `[§]rep-to=2;msg="news"[§]…;mach-status="ready";…;working=0[£]`

⇦ `[§]rep-to=2[£]` *(if function not supported)*

If the reply is positive but empty (without a body), or negative, the function is not supported (versions prior to `2022-06`) and the client will have to choose a proper frequency for the *polling* described in the previous section.

> ⚠ As of now it is not possible to subscribe to the changes of fields not included in `$status`:
> ⇨ `[§]id=2[§]$subscribed="$status,added1,added2"[£]`
> But it's always possible to ask **MACOTEC** to add a certain field if it is deemed useful.

If the subscription was successful, the positive reply will contain all the values of the status fields in order to initialize them appropriately; from now on, the machine will send notification messages on its own initiative when one or more of the monitored fields change, for example:

⇦ `[§]msg="news"[§]step=4;msg-list=1,4,32[£]` *(changed step and active messages)*

It's recommended to always try to subscribe to status change to improve responsiveness and reduce the messages processing and data traffic related to the full-status polling.

> ℹ Successful or unsuccessful subscription to status change could only influence the frequency polling of the full status (ex. `5 s` if subscribed and `0.5 s` if not)

> ⚠ It is better not to entirely give up on polling the full status, as this allows for the detection of potential server crashes or failures.

Its very important to handle notifications of machine disconnection, which are of the type:

⇦ `[§]ret=5;msg="news"[£]`

So that any (now unreliable) status information is removed from the graphics until the connection is restored.

To unsubscribe from notifications, simply redefine the `$subscribed` group with an empty string:

⇨ `[§]id=538[§]$subscribed=""[£]`

⇦ `[§]rep-to=538[£]`

> ℹ Closing the connection implicitly cancels the subscriptions, which must be renewed with each new connection.

> ℹ Clients who subscribe to status change notifications are excluded from inactivity checks (which result in disconnection after a certain timeout).

3.10.4 Common monitored fields

The most important common monitored fields are:

`can-receive-job`	Machine ready to get a new job
`job-loaded`	Machine has valid job data
`working`	Machine is processing glass
`scheme`	Current *Scheme* index (`1..N`)
`scheme-done`	*Scheme* was fully processed
`emg-list`	List of active emergencies
`msg-list`	List of active messages
`mode`	Current working mode
`spdovd`	Speed modulation percentage (*speed override*)

Please refer to `Interface.xml` file for the description, meaning and interpretation of the fields.

Current emergencies and messages should be appropriately displayed; for both, the machine passes a sequence of integers; the string and related image should be resolved via an alarm description file (`Alarms.xml`) for the particular machine.

Special emphasis should be given to any `emg-list` items: if it is not empty, the machine is in emergency and cannot work.

There is a subset of fields that makes sense to monitor only during processing (`working=1`) and thus appear only during this state:

`scheme-progress`	Completion percentage of the current scheme
`scheme-remaining-time`	Time remaining to completion of the current scheme

For status display can be useful:

`mach-status`	Machine main status (*implementation-defined*)
`op-status`	Operation in progress (*implementation-defined*)

While the possible values of `mach-status` and `op-status` are guaranteed (see `Interface.xml`), the sequences and transitions may vary from machine to machine since they depend on their implementation: assuming, for example, that `mach-status` must always assume the `stopping` value before transitioning into `ready`, introduces an unnecessary dependency that makes the *HMI* not portable across machines.

See also machine-specific monitored fields in:

sect.7.8.1 Monitoring the Strato machine status below on p.78
sect.8.9.1 Monitoring the Float machine status below on p.110

3.11 Events notification

Starting from version `2025-06` of MacoLayer, the clients that subscribe to status change notification will receive also the machine events notification.
When a significant event occurs a notification `msg="event"` is sent, for example:

```
⇦ [§]msg="event"[§]event=user-button;user-buttons="start"[§]
⇦ [§]msg="event"[§]event=status-changed;generic-status=WORK[§]
```

For a comprehensive list of the available events, see sect.10.3 “Raw machine events” below at p.149

3.12 Generating a *Project*

A *Project* is essentially a work order, created by an *Optimizer* that from a set of available *Sheets* and a list of desired *Pieces* creates a sequence of *Schemes* (see sect. 2.2 “Some terms” above on p.11).

In this context, for conciseness, we will loosely use the term *Scheme* to refer to an aggregate that combines a *Sheet* of a certain material/format, a list of contained *Pieces* and the *Job* to obtain them.

Each *Scheme* is numbered with a consecutive index starting from one; for each of them corresponds a work file that contains a description interpretable by the machine.

For example, a *Project* consisting of three *Schemes* will produce three *Job* files in a certain temporary directory:

- `C:\MyHmi\Output\MachineName\`
  - `scheme-1.mac`
  - `scheme-2.mac`
  - `scheme-3.mac`

The content and format of these *files* is described in sect. 4 “Scheme descriptor format (mac)” below on p.36.

> The `mac` file extension stands for `mac`otec or `mac`hine and it’s used to identify the *Project* work files.
>
> The file name is arbitrary, but it is advisable to include the corresponding *Scheme* index, even though what truly matters is the value declared internally.

Housekeeping

It is advisable to always use the same temporary folder (possibly named after the machine) clearing each time its content, in-fact it is recommended, when transmitting a new *Project*, to delete all previous `mac` files without any fear: they do not need backups because they can be regenerated at any time.

> It is the responsibility of the external application to clean the files it produces, not only to avoid filling up the hard drive, but also to avoid the disaster of mixing *Schemes* of different *Projects*, which at best wastes material, damaging the glassworks.

> The work files could be deleted by the machine itself to compensate for the shortcomings of a sloppy program that don't delete them.

3.13 Project transmission

The *Project* must be re-transmitted whenever it's necessary to confirm that the correct *Job*, coherent with the *Project*, is uploaded/synchronized to the machine. For example when a new *Project* is opened (ex. program startup) or when modifying an existing one.

Once the `*.mac` files have been generated in a certain folder, the *Project* is submitted to machine with the command:

⇨ `[SOH]id=1432[STX]prj-name="demo1";prj-path="C:\MyHmi\Output\MachineName"[ETX]`
⇦ `[SOH]rep-to=1432[ETX]`

> This may be a lengthy operation: implement user feedback, like a loading animation, to ensure a smooth experience.

> The machine will search for `*.mac` files in the folder specified in `prj-path`; unexpected files may cause an error.

3.14 Serving prints

The *HMI* must be ready to receive any incoming requests, particularly those that involve printing one or more piece labels:

⇦ [§]id=1433[§]prj-name="demo4";scheme=2;print-labels=12,1,14,3;printer="Avery 5.4 300dpi"[¶]

In this example, the machine requests the transmission of prints by specifying a sequence of four labels related to a certain *Scheme* of a certain *Project*.

> ⚠ The sequence of the requested label indexes must be maintained, ensuring that the print queue and the order in which labels are applied remain consistent.

> ℹ If `prj-name` or `scheme` are missing, assume them equal to the current ones.

At this point the *HMI* resolves the specified *Scheme* descriptor and the label identifiers; if it does not find all the information required for the print it will respond with an error, for example:

⇨ [§]rep-to=1433;ret=6;msg="demo4 scheme 2 does not have label 14"[¶]

> ℹ In this example is returned the error code `6` (`RET_VAL_UNAVAIL`) because the requested data was not found, however the important thing is that it is greater than zero.

Otherwise the *HMI* will proceed to create and transmit the required prints, in the specified sequence, to the printer indicated by `printer`; if not indicated, it will use the system's default printer or the one defined in its own configuration, if provided.

When the print transmission is complete (successful calls to the Print *API* and empty print queue), the *HMI* will give a positive response:

⇨ [§]rep-to=1433;msg="printed 12,1,14,3"[¶]

> ℹ The `msg` field is not mandatory but is useful for diagnostics.

Or negative in case of a printing error, for example:

⇨ [§]rep-to=1433;ret=8;msg="failed to print label 3: timeout expired"[¶]

> ℹ In this example is returned the error code `8` (`RET_FAILED`) because the requested action could not be performed, however the important thing is that it is greater than zero.

3.15 Editing material parameters

The material processing parameters are a set of quantities (pressures, speeds, times, …) used to cut a certain glass of a given thickness and characteristics on a certain machine.

If the machine independently manages the material parameters (see `using-internal-materials-db`), the *HMI* does not have to implement any editing dialog: when the operator request to edit the material parameters, the *HMI* will simply ask the machine to handle it:

⇨ [§]id=1435;msg="edit-material"[¶]

⇦ [§]rep-to=1435[¶]

This will open a dialog to edit the last synchronized material; it's also possible to specify the identifier of the material to edit:

⇨ [§]id=1436;msg="edit-material"[§]glass-id="44.1"[¶]

⇦ [§]rep-to=1436[¶]

3.16 Machine commands

With the field `machine-command` is possible to dispatch a direct command to the machine.

Command	Description
`start`	Start processing
`stop`	Abort operation
`reset`	Reset emergencies/status

Example:

⇨ `[§]id=2021[§]machine-command="stop,reset"[§]`

⇦ `[§]rep-to=2021[§]`

3.16.1 Requesting a start

Due to the delicate nature of the operation, special attention must be paid to the `start` command, equivalent to pushing the start processing button on the machine.

Machines are equipped with barriers and laser scanners to ensure that no harm will be done if the operator is in an unsafe zone. However, safety involves multiple layers, one of which is carefully following these instructions:

1) Just before requesting a start, ALWAYS verify that the *Job* to be executed is the intended one
2) If something fails, NEVER retry. Instead, notify the operator to press the start button manually

Here's an example. First check what is starting:

⇨ `[§]id=2340[§]job-loaded;prj-name;scheme;glass-id[§]`

⇦ `[§]rep-to=2340[§]job-loaded=1;prj-name="myprj";scheme=2;glass-id="44.1"[§]`

⚠ In case of a *Strato* machine, verify also the value of `step`.

If the *Job* on the machine doesn't match, abort and notify. Otherwise proceed with:

⇨ `[§]id=2341[§]machine-command="start"[§]`

⇦ `[§]rep-to=2341[§]`

In case of negative reply:

⇦ `[§]rep-to=2341;ret=7;msg="permission denied"[§]`

Or if after a reasonable time (`1 s`) in the monitored status `working` remains `0`, abort and notify the operator to start the processing himself.

3.17 Job desynchronization

There are situations where it's recommended to ensure that the machine does not have a valid *Job* uploaded/synchronized, to prevent any undesired action when the start button is pressed.

Typically, when the *HMI* closes, it is good practice to clear any transmitted *Job* from the machine.

The message to request the machine to clear any uploaded/synchronized *Job* is simply:

⇨ `[§]id=1435[§]mode="manual"[§]`

This will be interpreted as a "*desynchronization*": the effect is to invalidate the selected *Job*, thus inhibiting any machine actions if the start button is pressed.

If the *HMI* provides the user with an explicit command to desynchronize (ex. pressing the transmission button again), and the user triggers it while the machine is working, the actions to be taken can vary (possibly based on user configuration):

- [*default*] Proceed anyway with desynchronization as above
- [*only-if-not-busy*] Desynchronize only if the machine is not working (`working:0`), forcing the operator to stop the machine first (a notification would be appreciated)
- [*force-stop*] Request the machine to stop sending the following:
  `[§]id=1435[§]mode="manual"; machine-command="stop"[§]`

3.18 Keeping the connection alive

The communication channel closes automatically when the *client* closes the *socket* or after some time of inactivity (excluding *clients* subscribed to status changes). At a lower level, the *socket* itself may automatically close due to inactivity (see "*keepalive*").

Normal monitoring of the machine is sufficient to maintain the connection; in special cases there is the possibility of sending messages (ex. every minute) for the sole purpose of keeping the connection active:

⇨ `[§]id=2345;msg="ping"[§]`
⇦ `[§]rep-to=2345;msg="2017-11-06 11:33:32.359"[§]`

The *ping* message can be useful for synchronizing clocks in case the *client* is run remotely.

3.19 Disconnection

Even though it's not mandatory, the protocol allows the *client* to send an explicit disconnection notification:

⇨ `[§]msg="disconnect"[§]`

In case there's is time to wait for a reply (ex. if for some reason you disconnect to reset the connection), it's possible to send a request instead of a notification:

⇨ `[§]id=3432;msg="disconnect"[§]`
⇦ `[§]rep-to=3432;msg="bye"[§]`

4 Scheme descriptor format (mac)

4.1 Objectives

Describe the format of `mac` files that contain the processing information for a *Scheme*.

4.2 Context

In the past, *Job* data were expressed in terms of the machine specific implementation (*CNC* registers, G-code dialect, etc.), forcing the *HMI* to handle multiple outputs depending on the particular machine: this had many disadvantages, the tight coupling makes the overall system fragile and cumbersome to change.

The new approach is to hide machine internal details and let the *HMI* generate a generic descriptor of the *Job* that remains essentially the same for past and future machine implementations.

A *Project* consists of a set of *Schemes* to be processed, where each *Scheme* is described with a text file: this chapter specifies its format.

On how submit `mac` files to the machine, see sect. 3.12 “Generating a Project” above on p.32.

4.3 Job structure

The choice to have a separate descriptor for each *Scheme* is motivated by the desire for a clean division of the overall *Project* into completely independent units (*Schemes*), physically referable to a single file.

This has some advantages, such as more flexibility in the transmission mechanism (ex. creation *on-the-fly* and transmission of the single *Scheme* on-demand) and having less data to focus on when troubleshooting.

The disadvantage is replicating in each unit some of the data shared in the *Project*, (finished *Pieces* data and *Shape* definitions) but this is a small price to pay for greater data modularity.

4.4 Syntax

4.4.1 Encoding and syntax

- Text file encoded as `UTF-8` (without `BOM`)
- The syntax is a sort of MS `ini` extended with `json`[5]
- The parser is in general *case sensitive*, so respect the case of fields specified in this document (which is normally *lowercase*)
- Formatting spaces are ignored, line terminations are ignored inside `json` blocks, so field definitions can span between multiple lines
- String literals
  - Enclose in double quotes '"' (`U+0022`)
- Numeric literals
  - Decimal separator forced to '.' (`U+002E`) *dot*, regardless of local settings, for portability
  - No thousand separator supported
- Special reserved characters
  - '[' ']' (`U+005B`, `U+005D`) *square brackets*: `ini` section
  - '=' (`U+003D`) *equal sign*: `ini` key/value separator
  - '"' (`U+0022`) *double quotes*: string delimiter
  - '\' (`U+005C`) *backslash*: escape sequence or line continuation
  - ';' (`U+003B`) *semicolon*: `ini/json` line comment
  - '{' '}' (`U+007B`, `U+007D`) *curly brackets*: `json` block delimiters
  - ',' (`U+002C`) *comma*: `json` fields separator
  - ':' (`U+003A`) *colon*: `json` key/value assignment
  - '|' (`U+007C`) *vertical bar or pipe*: indicate span, range or aggregate
  - 'x' (`U+007C`) *Latin small letter x*: separate size values
  - '(' ')' (`U+0028`, `U+0029`) *round parentheses*: tuple delimiters

4.4.2 Values

Strings and double quotes

If a string literal might contain spaces or commas, it is necessary to delimit it between double quotes ('"' `U+0022`).

To insert double quotes in quoted string literals, escape them with the backslash ('\' `U+005C`): `key="blah \"inner quote\""`. This involves the necessity to escape the '\' character itself, especially if placed at the end of the string: `"blah\\"`.

Tuples

By delimiting a value in round brackets, the comma can be used to separate internal values such as the coordinates of a point: `(100,100)`.

5 For details see sect.11.2 “Extended json-ini syntax” below on p.154

4.5 Structure

The descriptor is divided into `ini` sections:

- `[project]` Information regarding the *Project* to which this *Scheme* belongs
- `[options]` Local options and units
- `[scheme]` About this *Scheme* and processing
- `[pieces]` Information regarding the obtained *Pieces*
- `[labels]` Information about the labels to be printed
- `[shape]` *Shape* descriptor
- `[steps]` List of processing *Steps* (*Strato* machines)
- `[paths]` Description of processing *Paths* (*Float* machines)

> ⚠ Maintaining the indicated order of these sections is advisable, although the only constraint strictly required is that the `[steps]`/`[paths]` section follows all others.

4.5.1 `[project]` section

Here you specify data regarding the *Project* from which the file was generated: its name, the *Scheme* index (`1..N`), and how many *Schemes* compose it.

- `name:<string>` (*project name*)
- `scheme:<int>/<int>` (*current scheme index/total schemes*)

4.5.2 `[options]` section

Optional section, contains possible directives to the importer, particularly how the numbers representing size or coordinates are to be interpreted; by default, they are interpreted in [`mm`].

4.5.3 `[scheme]` section

Contains data about the *Scheme*: *Sheet* size and material, type (*Strato*, *Float*, *Stripes* breakout, labeling, …).

The recognized fields are:

- `glass-id:<quoted string>` (material unique identifier)
- `glass-type:<stringlist>` (material type: `tpf,lowe`. Omit if standard glass)
- `size:<double>x<double>x<double>` (sheet size: *width*×*height*×*thickness*)
- `sheets:<int>` (*Scheme* multiplicity/repetitions)
- `type:<stringlist>` (generation criteria – Specifies the purpose for which the *Scheme* was generated)

`strato`	for *Strato* machine
`float`	for *Float* machine
`stripes`	for *Stripes* breakout
`labels`	for labeling *Pieces*
`no-lowe`	exclude low-E deletion

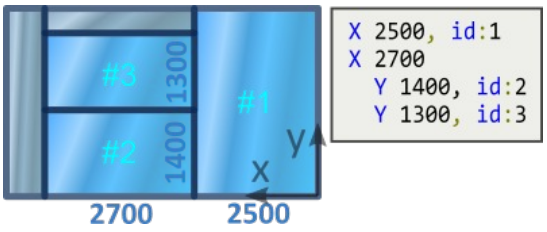
This is followed by the description of the cuts tree with XYZ notation (see 2.3 “Cuts tree/Scheme basics” above on p.12): each level specified with the letters `X,Y,Z,W,…` followed by the cut dimension and possibly by the finished *Piece* identifier.

First-level cuts (`X`) are assumed to be vertical. *Sheet* trims are not normally included.

- `(X|Y|Z|W|V|A|B|C):<string>` (*Scheme* cuts tree)

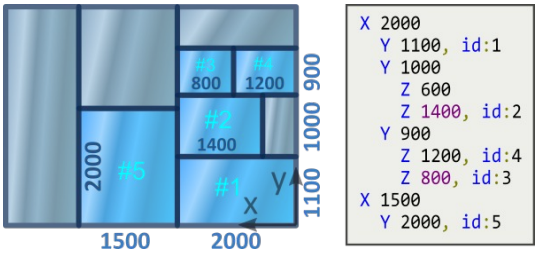
4.5.4 Cuts tree

Given a reference system and the dimensions of the *Sheet*, the guillotine cuts contained within it are described as shown in this example, where there are two first-level sibling cuts (`X`), the second of which (`X 2700`) has two children (`Y`).

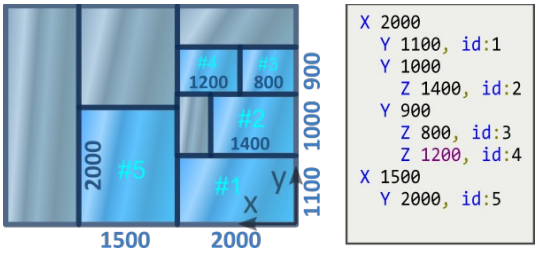


Equivalence

The description of the cuts tree is ideally independent of the type of machine and generally you can't assume that it will be strictly respected when processing the glass. Consider the two examples shown alongside: in the first case, the strict description involves the notation `Z 600` to describe the scrap encountered first along the positive direction of the X-axis, just as along this direction piece `#4` precedes piece `#3`.



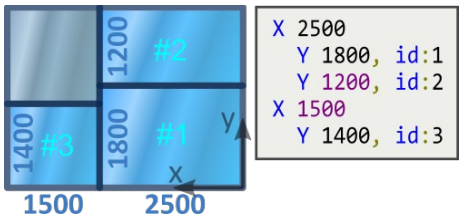
This description is probably unnecessary: the actual processing will depend on machine's characteristics; in this descriptive context, it may be advisable to consider the two *Schemes* equivalent and adopt the simpler notation.



Another way to look at it is to consider the positive direction of the axes dependent on the nesting level of the cut (`Z`,`W` inverted respect `X`,`Y`).

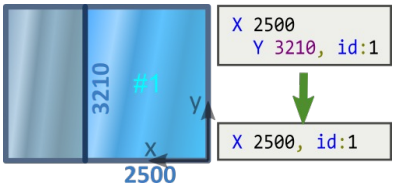
Virtual (containing) cuts

As shown in the previous examples, it is possible to associate any descriptors (ex. piece identifier) to the cut, referring to the *Product* of the cut itself. Consider this example, where piece `#2` is not the *Product* of a cut but rather the *Remnant* of a previous cut: to define it, a virtual cut `Y 1200`, coinciding with the edge of the sheet, is inserted solely to define a *Product* to which associate the piece descriptor. Similarly, the virtual cut `X 1500` is inserted to introduce the child cut `Y 1400` that obtains piece `#3` from the sheet *Remnant*. The virtual cuts are, of course, not executed: they are only a notational device to refer to the *Remnant* of the previous cut.

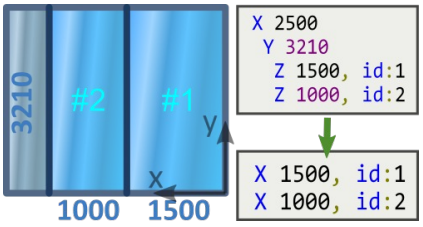


Normalization

In the example shown alongside the two notations are equivalent, but only the latter is normalized: the former defines a virtual cut `Y` that is in fact avoidable.



In this example too, the two notations are equivalent, where the former is normalizable. This kind of 'exotic' notation can be useful to highlight a cutting sequence different from the one obtained through the normal *tree preorder traversal*: in fact, the first case suggest to cut first at `X 2500`, while the typical sequence would be `X 1500` followed by `X 1000`.



4.5.5 `[pieces]` section

Contains information regarding the obtained pieces, sorted by piece identifiers (unique indexes).
The fields recognized in the part descriptor are:

- `size:`*`<double>`*`x`*`<double>`* (piece size *width*×*height*)
- `label:`*`<int>`* (assign a label index; as default the assigned label has the same piece index 0:no assigned label, eg don't print anything)
- `lowe:`*`<double>`*`,`*`<double>`*`,`*`<double>`*`,`*`<double>`* (low-E width on left, top, right, bottom sides)

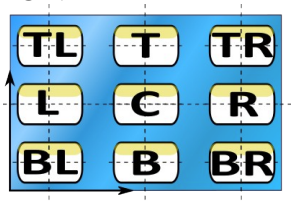
It is encouraged to add any other arbitrary additional fields; here is a non-exhaustive list of recognized ones:

- `code:`*`<string>`* (additional identifying code)
- `group:`*`<string>`* (belonging group)
- `shape:`*`<string>`* (information on contained *Shape*)
- `customer:`*`<string>`* (information about the client)
- `order:`*`<string>`* (about the order)
- `descr:`*`<string>`* (description)
- `note1:`*`<string>`* (custom note 1)
- `note2:`*`<string>`* (custom note 2)
- `note3:`*`<string>`* (custom note 3)
- `date:`*`<string>`* (piece shipping date)
- `rack:`*`<string>`* (information about destination)
- `edging:`*`<string>`* (edge grinding -finished piece will be smaller-)
- `work:`*`<string>`* (information on subsequent job on piece)

4.5.6 `[labels]` section

Contains information regarding piece labeling, each entry has as key the index of the associated *Piece* of `[pieces]`.
The recognized fields are:

- `label-size:`*`<double>`*`x`*`<double>`* (*optional*: label paper size *width*×*height*)
- `alignment:`*`<string>`* (*optional*: label positioning inside piece, possible values are: `TL`, `T`, `TR`, `L`, `C`, `R`, `BL`, `B`, `BR`)



The fields recognized in the descriptor are:

- `pos:(`*`<double>`*`,`*`<double>`*`)` (*optional*: placement coordinates of label center) accepted also `X:`*`<double>`* and `Y:`*`<double>`*
- `piece:`*`<int>`* (*optional*: override associated *Piece*, set to `0` to exclude the label)
- `print:`*`<string>`* (*optional*: path to print meta-file)

4.5.7 `[shape]` section

Contains the description of a *Shape*: for the format see sect. 5 “Polygonal shapes descriptor” on p.42 and sect. 6 “Generic shape descriptor” on p.52.

4.5.8 `[steps]` and `[paths]` section

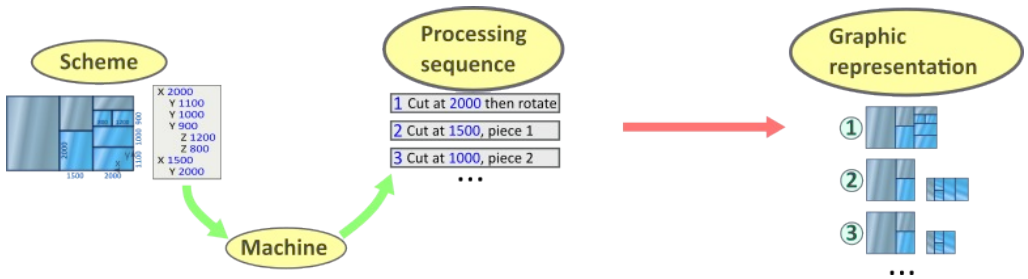
The content of these sections depends on the characteristics of the specific machine; they follow the common sections described above and contain how the *Scheme* is processed.

The `[steps]` section defines the *Step* sequence performed on *Strato* machines to process the *Scheme*, for details see sect. 7.9 “Strato Scheme descriptor (mac)” below on p.86.

The `[paths]` section defines the paths of each tool needed to process the *Scheme* on *Float* machines, for details see sect. 8.10 “Float Scheme descriptor (mac)” below on p.121.

These contents can be specified to give the *HMI* maximum control over the *Job*. However, they can theoretically be generated by the machine itself from the previous data, allowing for a simplified mode where the *HMI* completely omits these sections, leaving the *Job* generation to the machine.

To support this mode, the *HMI* should be able to recreate the graphics representing the work from an external *Job* sent back from the machine.



4.5.9 Examples

Examples of `mac` files can be produced through the `TestBoard` window in MacoLayer. (see sect. 11.1.2 “Getting started” below on p.152).

This document contains examples of `mac` files starting from sect. 7.9.4 (p.90) for *Strato* machines and starting from sect. 8.10.12 (p.131) for *Float* machines.

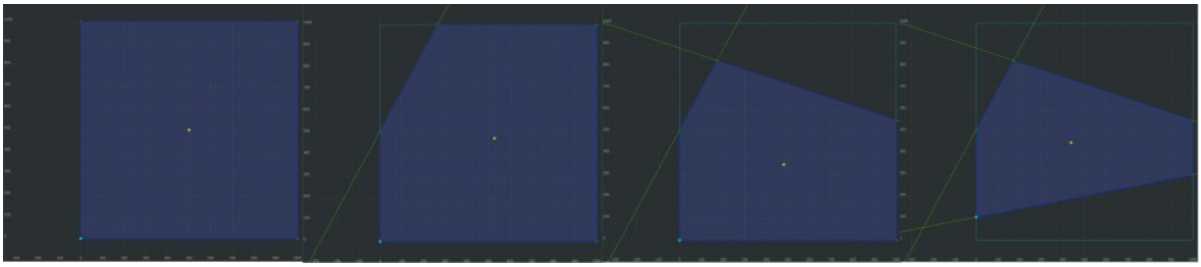
5 Polygonal shapes descriptor

5.1 Objectives

Specify the format to describe a convex polygonal *Shape* in terms of straight oblique (slant) guillotine cuts applied to a primitive glass rectangle.

5.2 Context

MACOTEC *Strato* glass cutting machines equipped with a grabber that can rotate glass[6] can perform slant cuts at arbitrary angles to obtain *Shape*s that can be described by convex polygons:



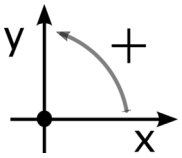
This family of *Shape*s can be operationally described by the sequence of cuts that produce them.

6 `mach-capabilities` contains `rotate` (see sect. 7.6 “Strato machine characteristics” below on p.75)

5.3 Concepts and terms

5.3.1 Reference system

To simplify, the following considerations assume a right-handed Cartesian coordinate system, where positive angles are counterclockwise.



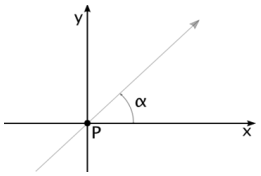
Of course, the same considerations can be applied using a left-handed system by reversing the direction of the x-axis and the terms *right*/*left* and *clockwise*/*counterclockwise*.

5.3.2 Cuts orientation

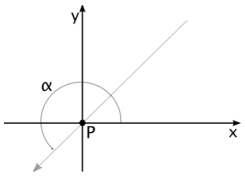
Cuts are oriented, meaning they are characterized by an oriented line, relative to which the angle formed with the x-axis is measured. The orientation of a line is determined by the sign of the coefficients of its equation in implicit form:

$$A \cdot x + B \cdot y + C = 0$$

Considering, for example, the line passing through the origin $y=x$ we have two possible representations:



$-x+y+0=0 \quad (A=-1, B=1, C=0)$
$arg(-A, B) = arg(1,1) = 45°$



$x-y+0=0 \quad (A=1, B=-1, C=0)$
$arg(-A, B) = arg(-1,-1) = -135°$

The two equations describe the same locus of points but impart opposite orientations to the line: the first is inclined `45°` and points toward the positive half-plane, the second `-135°` (or `225°`) and goes toward the negative half-plane.

Another example, the Y axis ($x=0$) is represented by $A=-1, B=0, C=0$ and not by $A=1, B=0, C=0$, in fact it must be $arg(-A, B) = 90°$.

Each line divides the plane into two half-planes, given its orientation we can distinguish and identify the one to the left and the one to the right[7]; for example, the half-plane to the right of the Y axis is x>0.

5.3.3 Cross-cuts

In this context a *Cross-cut* consists of an oriented line that applied to an initial glass (original *Part*) divides it into two next-level *Parts*, children of the original *Part*.

We have to decide which of the two *Parts* is the one we want to obtain (*Product*) and which is what is a left over (*Remnant*); since the cut is oriented we can distinguish what remains on the right from the one on the left; by convention the *Product* is the *Part* that remains on the right[8].

A *Product* that has no more nested cuts is a *Piece* (a leaf of the cuts tree[9]).

7 More generally, any oriented path divides the plane into two zones, left and right. If the path is closed, one of the two zones will have a finite area.

8 This convention comes from considering 'straight' strato glass machines, where the *Align* area is to the right of the *cutting axis*, where the finished *Pieces* are usually located. This arbitrary choice is retained for backward compatibility but is not consistent with the conventions adopted for generic *Shapes*.

9 Similarly to the considerations made in sect. 2.3 “Cuts tree/Scheme basics” above on p.12.

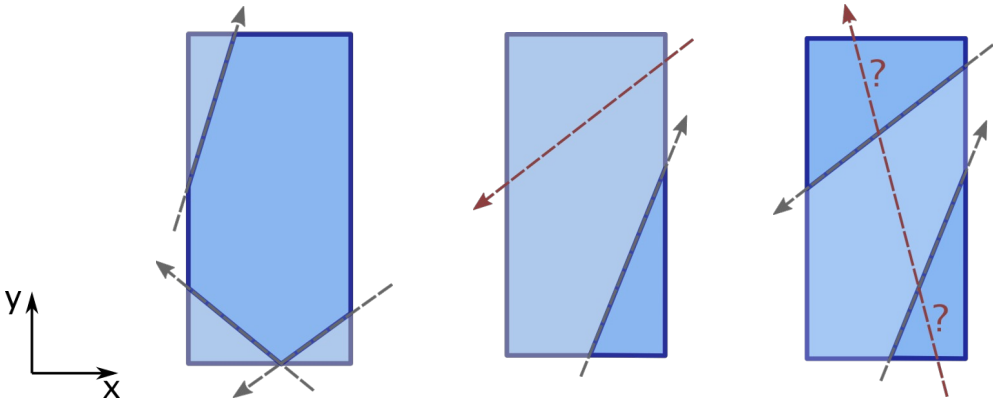
5.3.4 *Shape* primitive rectangle

The term "primitive piece/rectangle," in short *Primitive*, indicates the rectangular glass piece that contains the *Shape*; in other contexts may be called "axis-aligned bounding box" or "box".

5.3.5 Cuts tree and processing sequence

In the usual glass processing, cuts are lines parallel to the axes and applied to axis aligned rectangles; the natural generalization is having cuts given by generic lines applied to convex polygons.

Here are some examples to clarify the concept:



In the figure on the left, each cut can be applied to the *Product* of the preceding, in other words the *Products* of each cut have a non-null intersection, the finished *Piece*. Note that the resulting shape is independent of the order in which the cuts are applied.

A more interesting situation is exemplified in the figure in the center, where the sequence of application of the cuts matters: there is a cut that does not intersect the *Product* of its predecessor: the *Products* of the cuts are disjointed, consequently one cut invalidates the subsequent cuts, or rather, the cuts are applied to different *Parts*. This situation is not ambiguous because it is sufficient[10] to establish that the user wants to obtain two finished pieces from the initial *Primitive*, and it's possible to establish to which *Part* the cuts are applied.

In contrast, the figure on the right exemplifies an ambiguous situation: to which *Part* the third cut (in red) is applied? It needs additional information to determine the correct sequence of cuts and to which *Parts* they should be applied.

The processing sequence must be reduced to a list of items that reflect the situation in the figure on the left, consisting of an initial *Part* and the cuts applied to it. This list is obtained by appropriately serializing the cuts tree.

5.3.6 Summary

A *Cross-cut* is an oriented line applied to a finite area of the plane; it creates two *Parts*, the one on the right is the *Product* of the cut, and the one on the left is its *Remnant*.

Given a *Primitive* and a sequence of *Cross-cuts*, ambiguous situations may arise that require additional information to determine to which *Part* the cuts are applied; in such cases, it becomes important to declare a sequence and name the resulting *Parts*.

10 Unfortunately, it is not also necessary.

5.4 Devising the descriptor

To avoid the ambiguous situations mentioned above it is necessary that the descriptor contains information regarding the sequence of cuts and to which *Part* they should be applied.

We have seen that all we are doing is generalizing the usual glass processing: let's revise the commonly adopted descriptor[11]:

```
; Initial Part (Axis Aligned Bounding Box)
size=6000x3210

X=1000 ; First cut (level 1)
  Y=500 ; (level 2) Applies to the product of parent X=1000
    Z=450 ; (level 3) Applies to the product of parent Y=500
X=2000 ; (level 1) Applies to the remnant of sibling X=1000
  Y=1100 ; (level 2) Applies to the product of parent X=2000
  Y=600 ; (level 2) Applies to the remnant of sibling Y=1100
; ...
```

This descriptor fully describes the cuts tree, also suggesting a processing sequence.

As for which *Part* the cut is applied to, the rule is very simple: the first cut is applied to the *Primitive*, the following cuts are applied according to their level, either to the *Remnant* obtained from the previous sibling cut or to the *Product* obtained from the parent cut.

For our generalization we borrow the same concept, without inventing anything new; the only difference will be the notation of the cuts (which being generic lines will be a little more complex) and in declaring the level with a number instead of a letter, mainly to avoid unnecessary limitations, but also because these letters lose meaning in our generalization.

11 See also sect. 2.3 “Cuts tree/Scheme basics” above on p.12.

5.5 File format

5.5.1 Encoding and syntax

- Text file encoded as `UTF-8` (without `BOM`)
- Syntax similar to MS `ini` extended with `json`[12]
- Fields and attributes names are all *lowercase*: it is recommended to respect this because the parser may be case sensitive
- Spaces are generally ignored, line breaks are ignored within `json` blocks allowing definitions on multiple lines
- Numeric literals
  - Forced decimal separator to '`.`' (`U+002E`) *dot*, independent from local settings
  - No thousand separator supported
- Reserved characters
  - `[ ]` (`U+005B`, `U+005D`) *square brackets*: `ini` sections
  - `=` (`U+003D`) *equal sign*: `ini` field-value separator
  - `"` (`U+0022`) *double quotes*: string literals
  - `\` (`U+005C`) *backslash*: escape sequences and line continuation
  - `;` (`U+003B`) *semicolon*: `ini` line comment
  - `{ }` (`U+007B`, `U+007D`) *curly braces*: `json` blocks delimiters
  - `,` (`U+002C`) *comma*: `json` fields separator
  - `:` (`U+003A`) *colon*: `json` key-value assignment
  - `( )` (`U+0028`, `U+0029`) *round brackets*: tuple delimiters

5.5.2 Numeric quantities

- Coordinates and dimensions in `[length]`
  - Default is millimeters (`length:0.001`), otherwise declare `length` appropriately (ex. meters: `length:1`, centimeters: `length:0.01`, inches: `length:0.0254`)
  - Round to no less than 10^{-4}mm, extra decimals would be useless in glass cutting
- Angles in `[deg]`
  - For angle rounding we can cautiously set a maximum error of 1µm over 10m, in which case the angular fraction to which to round is 10^{-7} radians, or 10^{-6} decimal degrees. For this reason, notations without angular values are preferred

⚠ Rounding of angles has a distance-dependent effect, so it is generally preferable to avoid notations containing angles.

⚠ Decimal numbers should be represented in a portable (locale independent) way, using the dot as the decimal separator.

12 For details see sect.11.2 “Extended json-ini syntax” below on p.154

5.5.3 Fields of polygonal shapes descriptor

Header

Field	Value	Notes	Default
`name`	*`string`*	*Shape* unique name	

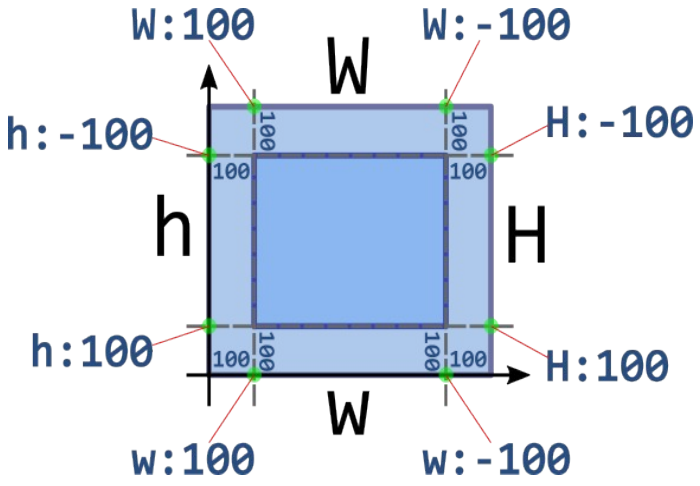
`options`	*`json sequence`*	Options list
		Field / **Type** / **Notes** / **Default**
		`strict` / *`int`* / `1`: strictly respect declared sequence and orientation
		`0`: use max area criterion to choose the proper orientation / `0`
		`length` / *`double`* / [m] Units used for lengths / `0.001`
			`{strict:0, length:1E-3}`

`size`	*`double`* x *`double`*	Size of primitive `widthxheight` [`length`] (also accepted: `width=`*`<double>`* and `height=`*`<double>`*)	`0x0`

Cuts

Field	Value	Notes
`[int]` *`(cut level number)`*	*`json sequence`*	Description of the cut, what is obtained and what it is applied to. It can be given by the composition of several entities: **Field** / **Description** / **Notes** `line` / oriented line / side:offset, side:offset (intersection points with the primitive box, orientation from the first to the second point) `to:(x,y)`, `from:(x,y)` (points) ~~`x1,y1,x2,y2` (points coordinates)~~ `from:(x,y)`, `a` (point and angle respect x-axis) ~~`x,y,a` (coords and angle respect x-axis)~~ `A,B,C` (coefficients of the implicit form A·x+B·y+C=0) `id` / *Part* identifier / A unique string to identify the *Product* obtained by this cut

Notation by primitive intersection points - side and offset



5.5.4 Format exemplification

```
[shape] ; section title
name="poly-shape-3" ; shape identifier, required
options={strict:0, length:0.0254} ; directives to the importer

; Axis-Aligned Bounding Box (initial rectangle)
size=50x100 ; [in]

; Cuts:

; Line description with bounding-box intersection points
1=line:{w:20.5 W:-10.2} ; side1:offset1, side2:offset2
; The bounding-box sides are:
;  y       +--W--+
;  ^      h|     |H
;  +->x    +--w--+
; offset is the distance from the side vertex
; (>0:from lower vertex, <0:from higher vertex)

; Line description with passing points:
2=line:{from:(0,0) to:(10.2,20.43)} ; (x1,y1)(x2,y2) [in]

; Line description with passing point and slope:
3=line:{from:(0,80.5) a:-45.213543} ; (x,y)<a° [in] [deg]

; Line description as equation in implicit form:
4=line:{A:-0.2 B:1.535532 C:-54.3} ; Ax+By+C=0

; Naming a product
1=line:{w:20 W:-10}, id:2 ; obtains piece 2

; Axis aligned cuts
X=30.2 ; A first-level vertical cut
  Y=15.1, id:3 ; obtains piece 3
```

5.6 Examples

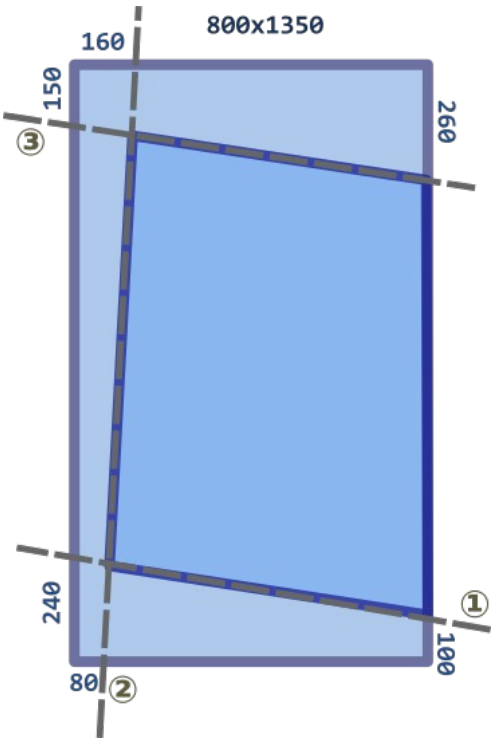
5.6.1 Single polygonal shape

This is the usual situation, where the sequence of cuts declaration is irrelevant, the important thing is that each cut is applied to the *Product* obtained with the previous one, so the level of cuts continues to increase.

If `strict:0` (the default) the interpreter will decide the cut orientation by applying the maximum area criterion (the *Product* of each cut is the *Part* with the largest area). Thus, it is possible to define the cutting lines without particular precautions and let the importer do the work.

In this example, the cutting lines are described through the points of intersection with the *Primitive* rectangle: this notation is convenient because it directly show the expected dimensions of the scraps. We can observe that if one wants to minimize waste, one side of the polygon will always coincide with a side of the *Primitive*, so:

- If you want to obtain a quadrilateral, it wouldn't make much sense to have more than three cuts
- Parallel cuts to the primitive make no sense: it would be better to simply reduce its size




```
[shape]
name="quadrilateral"
size=800x1350 ; [mm]

1=line:{H:100 h:240}
  2=line:{w:80 W:160}
    3=line:{h:-150 H:-260}
```

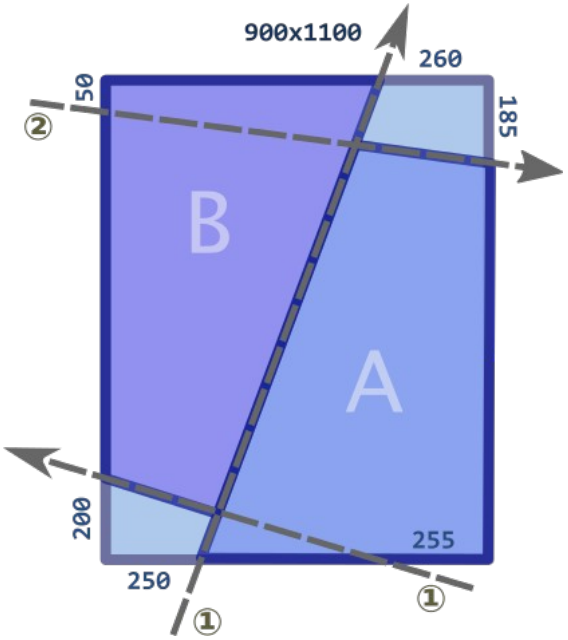
⚠ The number to the left of the equal sign is NOT an identifier of the cut: it simply indicates the level at which the cut is applied. In this example, we have three cuts nested within each other.

ℹ Fractional numbers do not appear in the examples only for ease of reading.

5.6.2 Nested polygonal shapes

Now let's look at a situation that the importer cannot manage autonomously, where the sequence and nesting of cuts matters. We want to obtain two *Shape*s named `A` and `B` from a single *Primitive*.

In these cases it is advisable inform the *importer* not to take arbitrary initiatives (`strict:1`), as it is now necessary to clearly specify the correct orientation and sequence of cuts.




```
[shape]
name="disambiguating multi-shape"
options={strict:1}
size=900x1100 ; [mm]

1=line:{w:250 W:-260} ; DX
  2=line:{h:-50 H:-185}, id:A ; A
1=line:{w:-255 h:200}, id:B ; B
```

We start with the main cut, the almost vertical one in the middle, which *Product* is the right *Part*, containing the finished piece `A`.

The next cut is the one at the top; it is declared at level `2`, so it is a child of the previous cut and is applied to the *Product* of the parent, obtaining *Piece* `A`.

The last cut, instead, is at level `1`, so it is applied to the *Remnant* of the preceding sibling, that is, the left part of the *Primitive*; the *Product* is the finished *Piece* `B`.

The proposed format supports cases of much greater complexity than this example; however, it is easy to imagine how these significantly increase the operator's work, nullifying the savings on waste: it is therefore advisable and appropriate to divide them into less complex *Shapes* on more *Primitives*.

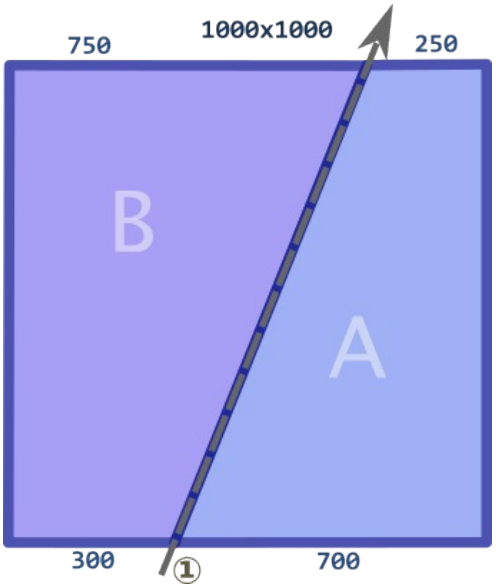
The descriptor seems to suggest a processing sequence, but do not assume that the machine will follow it: it's not the only possible one (the possible sequences are probably equal to the number of permutations of the output pieces, in this case `2!`).

The indentation in the file has no semantic value and is only for readability. The `id` attribute has also been added for the reader's convenience.

The option `strict:1` is specified to tell the importer to respect cuts orientations in the definition.

5.6.3 Indicating a *Remnant* as a finished piece

If the *Remnant* is not *Scrap* but a finished *Piece*, nothing changes operationally; however, it may be useful to assign it an identifier to somehow inform the operator that the *Remnant* is not to be discarded. To do this, simply add a *virtual* cut at the same level of the previous one (sibling), specifying only the identifier.




```
[shape]
name="naming remnant"
options={strict:1} ; trust me
size=1000x1000
1=line:{w:300 W:-250}, id:A ; Product
1=id:B ; Remnant is piece B
```

The option `strict:1` prevents the importer from swapping `A` and `B`, assigning `A` to the largest piece.

6 Generic shape descriptor

6.1 Objective

Specify the format to describe a generic *Shape* given by a series of paths formed by arcs and lines.

6.2 Context

Both MACOTEC *Float* and *Strato* machines can process generic *Shapes*; the latter can be equipped to move the glass along the x-axis interpolating it with the carriages.
A *Shape* is a collection of processing trajectories defined inside a rectangular *Piece*, called the *Primitive* of the *Shape*.

6.3 Concepts and terms

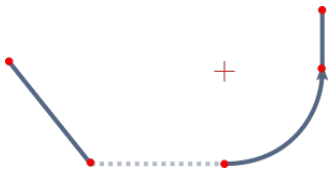
6.3.1 Paths, entities, shapes

The term *Path* refers to a continuous trajectory in the XY plane composed of a sequence of arcs/segments, which are referred to as *Entities*.
Entities are oriented and characterized by a destination point, given by the coordinates in the XY plane; since a *Path* is by definition continuous, the starting point of each entity coincides with the destination point of the previous one; the first entity starts at an initial point of the *Path*.
A *Path* is closed if the destination point of the last entity coincides with its initial point, and open otherwise.
The supported *Entities* are:

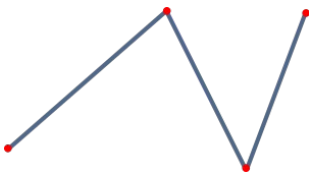
- Straight segments[13]
- Circular arcs

A *Shape* is given by the set of one or more *Paths*.

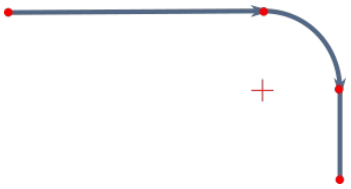
This is not one *Path*, but two:



This is a single *Path*, with discontinuities on the tangent (corner points):



This is a single *Path* whose tangent is continuous:



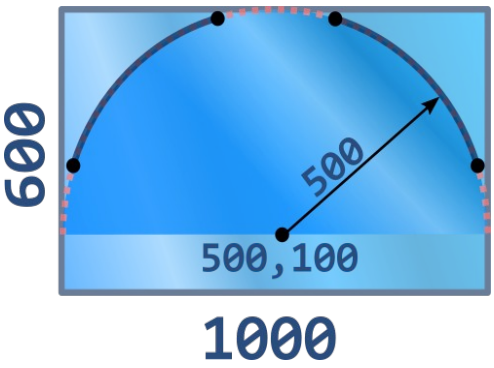
13 A straight segment can be thought of as an arc with an infinite radius.

6.3.2 Definition and actual processing

Do not confuse the *Paths* that define a *Shape*, hereafter referred to as *nominal*, with those that describe the actual processing with certain tools.

Consider the arched *Shape* in the figure beside: its *nominal* definition is a single *Path* consisting of a semicircle centered in `500,100` that connects `0,100` to `1000,100`. The *Paths* to process this *Shape* will be probably quite different to respect tool constraints.

For example, the scoring *Paths* must maintain a certain distance from the box edges and will likely be described with two arcs as outlined in the picture.



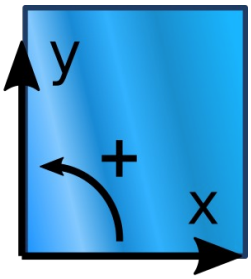
A *Shape* definition can generate different processing *Paths* depending on the tool used.

The processing of a *Path* that presents discontinuities on the first derivative (tangent) may involve tool re-orientations (tangent discontinuity threshold).

6.3.3 Reference system

The coordinates of the entities are relative to the *Primitive*, as depicted here. In the following examples we will assume a right-handed system, but all considerations also apply in case of left-handed systems.

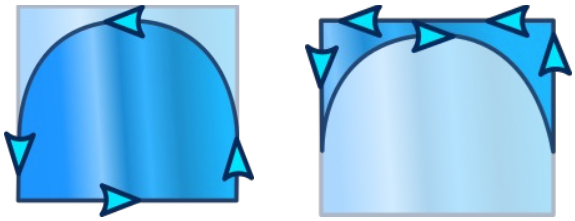
In any case, the positive direction of angles/arcs always goes from the x-axis to the y-axis: positive arcs are counterclockwise in right-handed systems and clockwise in left-handed systems.



6.3.4 Path orientation and inner area

Entities move towards their destination point and are therefore oriented: this can convey the information of where the "inside" of the *Shape* is (important information for correctly applying any offsets for tool multiple passes).

We define the "inside" of the *Shape* as being to the left of the *Path*, so *closed* shape definitions will have a path that overall will be counterclockwise (in right-handed coordinate system).



Be careful not to confuse the orientation of the *Entities* that define the *Shape* with the direction of the actual processing: they may not coincide.

Unfortunately, this choice is not consistent with the one arbitrarily set for polygonal *Shapes*, which is better left unchanged for backward compatibility; this minor inconsistency is annoying but has no consequences.

6.4 Format

6.4.1 Fields of generic shapes descriptor

Header

The fields in the header are quite similar to those introduced for polygonal *Shapes*:

Field	Value	Note	Default
`name`	*`string`*	*Shape* unique identifier	

`options`	*`json sequence`*	Options list
		Field / **Type** / **Note** / **Default**
		`strict` / *`int`* / ignored / `0`
		`length` / *`double`* / [m] Units used for lengths / `0.001`
		`origin` / *`double, double`* / Reference coordinates [`length`] / `0,0`
		`box` / *`double x double`* / Bounding box size [`length`] / `0x0`
			`{strict:0, length:1E-3}`

`size`	*`double x double`*	Size of primitive `widthxheight` [`length`] (also accepted: `width=`*`<double>`* and `height=`*`<double>`*)	`0x0`

Paths definition

Field	Value	Note

`[string]`
`(path attributes)`
	`json sequence`	`path`: Definition of a path, given by a sequence of entities:
		Field / **Description** / **Inner fields**
		`start` / initial point / coords: `from:(X,Y)`, origin: `origin:(X,Y)`, bounds: `box:WxH`
		`line` / generic line (segment or arc) / destination: `to:(X,Y)`, sagitta: `bulge:val`, tangent: `tan-diff:deg`
		`arc+` / positive arc / destination: `to:(X,Y)`, center (rel): `c:(X,Y)`, center (abs): `C:(X,Y)`
		`arc-` / negative arc

The coordinates of the arc center can be specified as absolute (uppercase `C`) or relative to the starting point (lowercase `c`).

The decimal separator must be strictly a dot: the comma is used as a field separator.

The *Shape* is considered invalid if one or more *Entities* are not contained within the primitive.

Be careful to correctly define the 'inside' of the *Shape*.

There is no concept of a closed *Path*: it must be explicitly closed with an *Entity* to is initial point.

6.4.2 Path attributes declaration: tools

The key to which the *Path* is assigned is a string of comma-separated items that represents its attributes, namely the processing tools.

This allows for a fine-grained description of the desired processing sequence, when necessary.

Key	Description
`*`	Tools not specified; they will be chosen automatically by the machine Use this when just defining the nominal *Paths* that define the *Shape*
`all`	All machine tools
`no-lowe`	All machine tools except the low-E wheel

`tpf,`
`lowe,`
`knife,`
`score,`
`open,`
`mark`
	List of tools, in order of priority

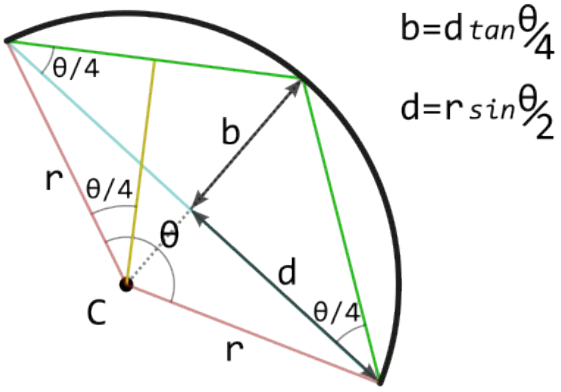
Tool name	Description
`tpf`	Temporary protective film removal scraper
`lowe`	low-E wheel
`knife`	Vinyl cutting blade
`score`	Scoring head(s)
`open`	Open/breakout tool(s)
`mark`	Marker pen

6.4.3 Curved *Entities*

The `line` field can represent an arc or a segment (which is still an arc, but of infinite radius).

To set the "curveness" it's possible to specify the *bulge*, aka the *sagitta* of the arc measured relative to the halved chord:

```
bulge = b/d = tan(θ/4)
```



If `bulge:0` (*default*) the *Entity* is a segment; if `1` is a positive semicircle, `-1` a negative semicircle, `2` a full circle, and so on.

Alternatively it is also possible to constrain the tangent at the starting point with `tan-diff`, expressed in decimal degrees, representing the relative angle of the tangent respect the previous (a value of `0` means tangent continuity, convenient when defining a smooth path without corner points), to see an example check "Forcing the tangent" below on p.59.

6.4.4 Nominal definition and processing

The descriptor of a generic *Shape* supports multiple levels of representation: it can depict either the conceptual outline (pure or *nominal* definition) or the actual operations required to process the *Shape*, the sequence of tools and their specific constraints (ex. distance from edges).

The first representation defines the *nominal* outline, while the second defines *actual* processing trajectories.

To understand the difference between the two descriptions, refer to the two examples below on p.58.

The `[shape]` sections in the `mac` file usually contain the *nominal* definition; the proper processing is generate elsewhere (in the `[paths]` section in case of *Float* machines or by the machine itself in case of *Strato* machines).

6.4.5 Bounding box

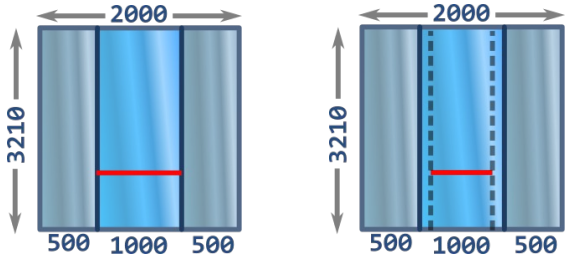
The *Paths* of the *Shape* must be contained within its *Primitive*, whose dimensions are specified in `size`.

It is possible to specify a more restrictive rectangular region by specifying in *options* the fields `origin` and `box`.

If `box` is specified, `origin` not only defines the relative origin respect with are referred the coordinates but also indicates the position of the bounding box reference corner (bottom-left or bottom-right depending on the reference system).

This is useful in case is necessary to apply the tool edges margins on a region smaller than the *Primitive*, for example the special processing of *Stripe* aggregates worked as generic *Shapes* on *Strato* machines.

For example consider this strange *Shape*, whose *Primitive* is artificially big:



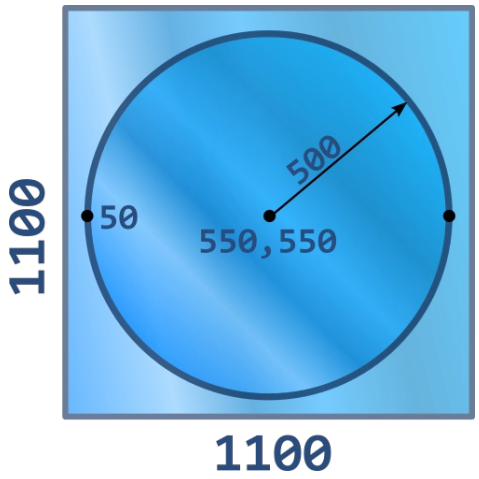

```
[shape]
name="aggregate-stripe-rects"
options={ origin:(500,0), box:1000x3210 }
size=2000x3210 ; Primitive
*=path:{
    start:{from:(0,700)}
    line:{to:(1000,700)}
   }
```

> ⚠ To ensure the bounding rectangle is correctly positioned, it is important that the `box` field is defined after `origin`.

Restricting the bounding box to the inner region ensures the proper margins for the *Shape Paths* (red line in the figure).

6.5 Examples

Full circle

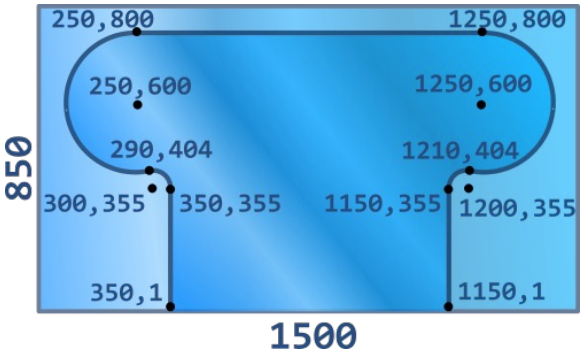



```
[shape]
name="circle_R500"
size=1100x1100
*=path:{
    start:{from:(1050,550)}
    arc+:{to:(50,550) C:(550,550)}
    arc+:{to:(1050,550) C:(550,550)}
   }
```

```
; Equivalent form:
*=path:{
    start:{from:(1050,550)}
    line:{to:(50,550) bulge:1}
    line:{to:(1050,550) bulge:1}
   }
```

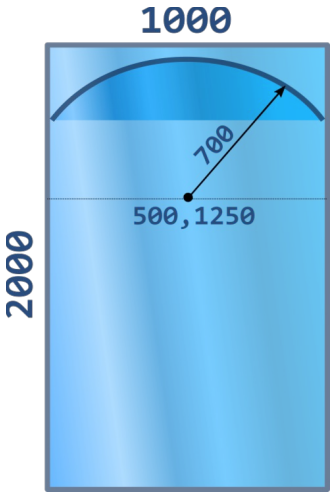
⚠ To avoid ambiguities the full circle is always defined with two semicircles.

Arcs and lines




```
[shape]
name="mushroom"
size=1500x850
*=path:{
    start:{origin:(0,0) from:(1150,1)}
    line:{to:(1150,355.051)}
    arc-:{to:(1210,404.041) C:(1200,355.051)}
    arc+:{to:(1250,800) C:(1250,600)}
    line:{to:(250,800)}
    arc+:{to:(290,404.041) C:(250,600)}
    arc-:{to:(350,355.051) C:(300,355.051)}
    line:{to:(350,1)}
   }
```

Arched rectangle

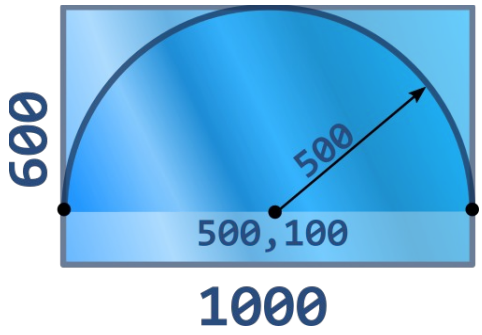



```
[shape]
name="arched_R700"
size=1000x2000
*=path:{
    start:{from:(999,1740.917)}
    arc+:{to:(1,1740.917) C:(500,1250)}
   }
```

```
; Equivalent form:
*=path:{
    start:{from:(999,1740.917)}
    line:{to:(1,1740.917) bulge:0.419}
   }
```

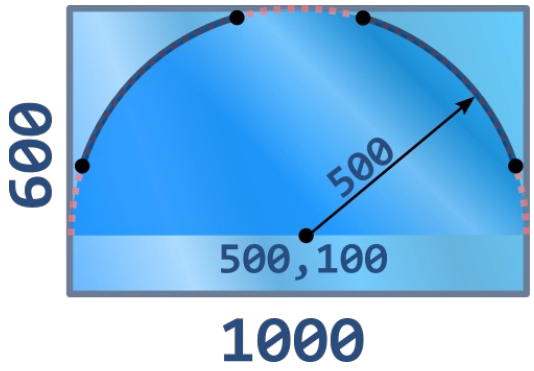
Specifying tool constraints

Nominal definition:




```
[shape]
name="semicircle-nominal"
size=1000x600
*=path:{
    start:{from:(1000,100)}
    arc+:{to:(0,100) C:(550,100)}
   }
```

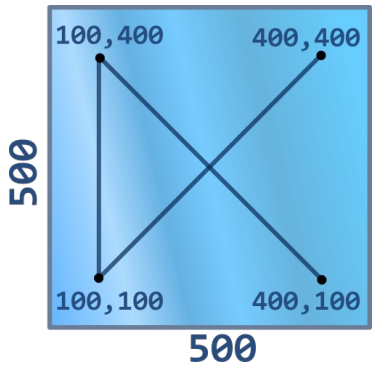
Forcing a distance of 1mm from edges:




```
[shape]
name="semicircle-actual"
size=1000x600
score=path:{
    start:{from:(468.4,599 )}
    arc+:{to:(1,131.6) C:(550,100)}
   }
score=path:{
    start:{from:(999,131.6)}
    arc+:{to:(531.6,599) C:(550,100)}
   }
```

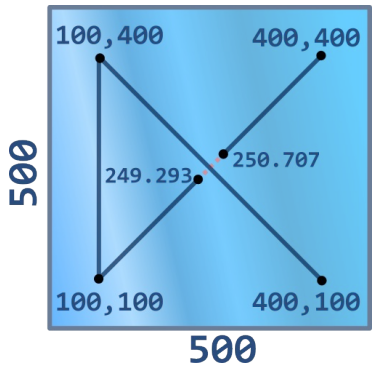
Intersecting cuts

Nominal definition:




```
[shape]
name="crossing-nominal"
size=500x500
*=path:{
    start:{from:(400,100)}
    line:{to:(100,400)}
    line:{to:(100,100)}
    line:{to:(400,400)}
   }
```

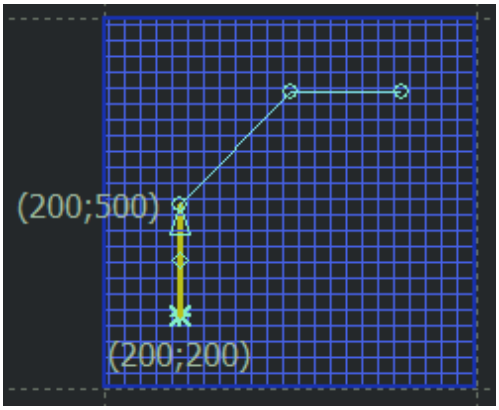
Forcing a margin of 1mm from crossing cuts:




```
[shape]
name="crossing-actual"
size=500x500
score=path:{
    start:{from:(400,100)}
    line:{to:(100,400)}
    line:{to:(100,100)}
    line:{to:(249.293,249.293)}
   }
score=path:{
    start:{from:(250.707,250.707)}
    line:{to:(400,400)}
   }
```

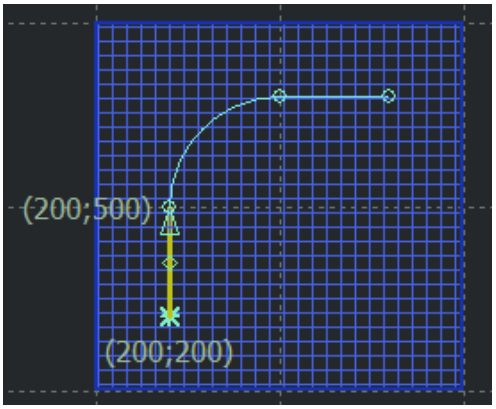
Forcing the tangent

Consider:



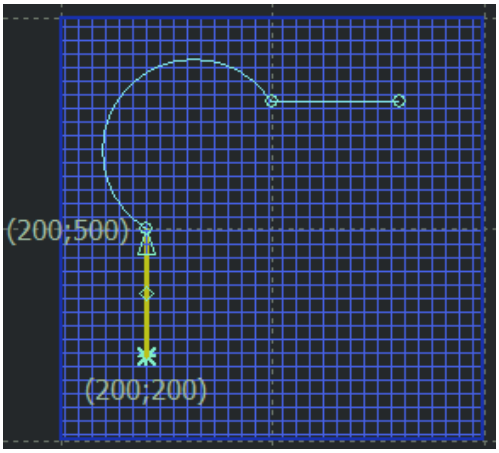

```
name="tangents-1"
size=1000x1000 ; [mm]
*=path:{
    start:{from:(200,200)}
    line:{to:(200,500)}
    line:{to:(500,800)}
    line:{to:(800,800)}
   }
```

Let us see what happens by constraining the tangents to have a certain angle difference. Forcing it to zero eliminates the corner points, smoothing the *Path* (continuity on the first derivative):



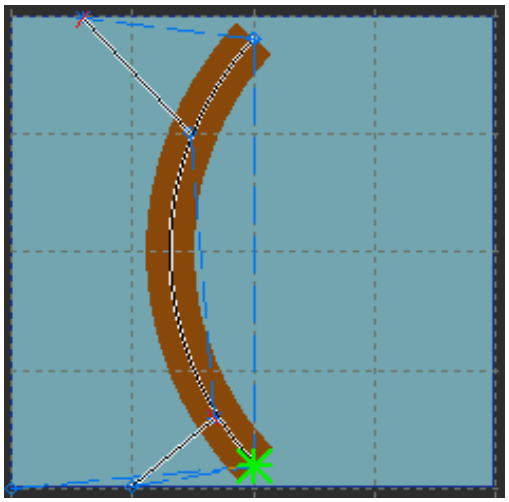

```
name="tangents-2"
size=1000x1000 ; [mm]
*=path:{
    start:{from:(200,200)}
    line:{to:(200,500)}
    line:{to:(500,800) tan-diff:0}
    line:{to:(800,800) tan-diff:0}
   }
```

Setting just for the fun of it a difference of `45°`:



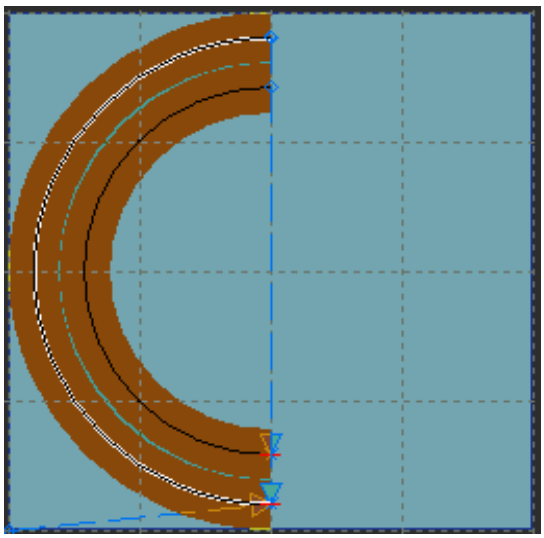

```
name="tangents-3"
size=1000x1000 ; [mm]
*=path:{
    start:{from:(200,200)}
    line:{to:(200,500)}
    line:{to:(500,800) tan-diff:45}
    line:{to:(800,800) tan-diff:45}
   }
```

Tools indication: additional score lines for breakup



```
[shape]
name="added score lines"
size=200x200
*=path:{
    start:{from:(100,190)}
    arc+:{to:(100,10) C:(200,100)}
   }
score=path:{
    start:{from:(30,199)}
    line:{to:(74,150)}
   }
score=path:{
    start:{from:(84,30)}
    line:{to:(50,1)}
   }
```

Tools indication: tpf+lowe multi-pass



```
[shape]
name="multi-pass"
size=200x200
tpf,lowe=path:{
    start:{from:(100,171)}
    arc+:{to:(100,29) C:(100,100)}
   }
tpf,lowe=path:{
    start:{from:(100,190)}
    arc+:{to:(100,10) C:(100,100)}
   }
score,open=path:{
    start:{from:(100,190)}
    arc+:{to:(100,10) C:(100,100)}
   }
```

7 *Strato* machines

7.1 Objective

Provide the concepts and criteria regarding the generation and processing of *Schemes* in our laminated glass cutting machines.

Clarify how the generic machine interface specializes for *Strato* machines: the specific resources and the content of the `mac` files.

7.2 Context

While older *Strato* machines needed to be externally driven by the *HMI* for each single cut, new machines can process an entire *Scheme* autonomously, drastically changing the *HMI* interaction required to process a glass *Project*.

For the basics of *Strato* machines, please refer to sect. 2.6 “Strato machines” above on p.14.

7.3 Reference systems

The machine's reference system is given by two orthogonal axes where the y-axis, transverse to the direction of the glass movement, constitutes the cutting direction, where the scoring heads move, and the x-axis, parallel to the direction of movement, has the positive half-plane in the *Align* area, where moving blocks square the glass aligning the cut with the scoring heads.

The *Scheme* reference system, the one to which the *Optimizer* and the *HMI* refers, has the origin on the reference vertex of the *Sheet* and has the x-axis direction inverted respect the machine reference.

In the so called "*straight*" machines, the operator sees the glass advancing from left to right, the machine reference system is right-handed and the *Sheet/Scheme* reference system is left-handed, while in "*opposite*" machines, is all inverted along the x-axis: the operator sees the glass advancing from the right to the left and the *Sheet/Scheme* reference system is right-handed.

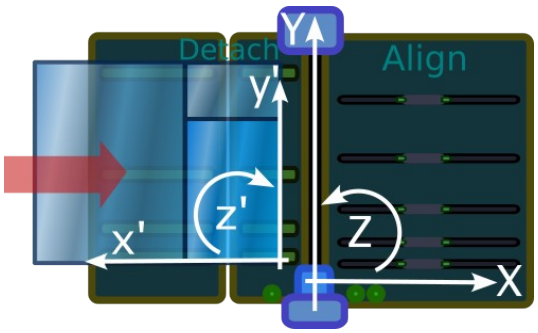


Figure 2: “Straight” Strato machine

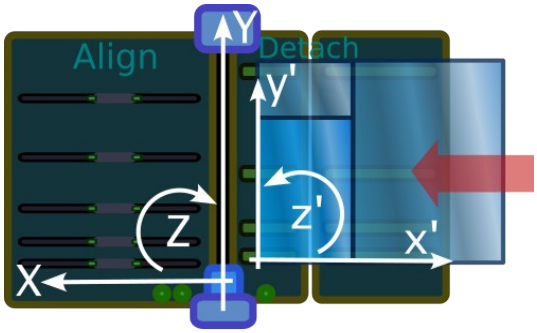
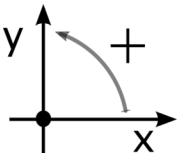


Figure 3: “Opposite” Strato machine

In a reference system, the positive direction of the angles coincides with the rotation that aligns the x-axis with the y-axis. In a right-handed system this sense is counterclockwise; and in a left-handed system clockwise.



Note that the *Sheet/Scheme* reference has the abscissas inverted respect the machine reference: a rotation that is positive for the machine is negative for the *Sheet*.

7.4 Schemes for *Strato* machines

As anticipated in sect. 2.3 “Cuts tree/Scheme basics” above in p.12, a *Scheme* is a set of cuts applied to a *Sheet* dividing it into smaller regions; the cuts are lines parallel to its sides. Each cut is applied to a rectangle of known dimensions. The cut traverses the rectangle from edge to edge[14], splitting it into two smaller rectangles; once all the cuts are applied, the rectangles to be obtained are called finished *Pieces*, while those remaining are called *Scraps*.

The *Scheme* is typically generated by an optimization algorithm, whose main criteria are to minimize waste (total area of the *Scraps*) and maximize the automatic processing on the machine.

The latter criterion depends on the machine's characteristics and is usually in conflict with the first; it becomes particularly significant in the case of fully automatic machines, to minimize the cost of operator's manual interventions. Typically, a generic algorithm creates the *Scheme*, which is then further processed to apply the machine-specific criteria.

7.4.1 *Scheme Job*: processing *Steps*

In order for a *Strato Scheme* to be processed, it must be transformed into a sequence of machining *Steps*. There are several possible *Steps* sequences that can process a certain *Scheme*, among these, there is one particularly suitable for being processed on a specific machine.

When generating the *Steps* sequence (*Scheme Job* generation) there’s two aspects to consider: how to choose the proper sequence and how to ensure the maximum workability on the particular machine (which we will address in the section 7.5).

7.4.2 Cuts sequence and cuts tree

Let's examine the criteria for selecting the appropriate processing sequence, starting with the choice of the sequence of cuts.

As anticipated in sect. 2.3 “Cuts tree/Scheme basics” above on p.12, given the nature[14] of the cuts, the most natural data structure to represent the work sequence is a binary tree.

First-level cuts are applied to the entire *Sheet*, and subsequent levels of cuts, their children, are applied to the *Subsheets* obtained from the parent cuts, and so on. The levels are clearly visible in the example in the adjacent figure, forming a hierarchy of cuts.

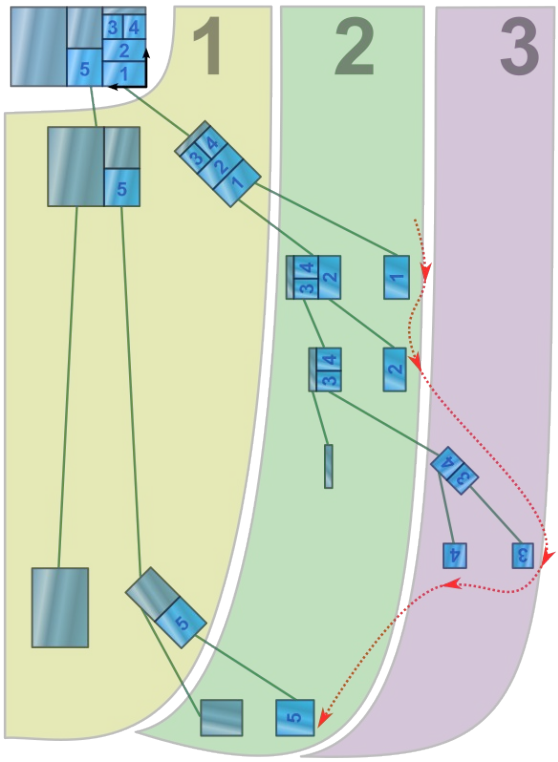


Figure 4: Cuts tree levels

Each cut divides the *Subsheet* to which it is applied into two *Parts*: *Product* and *Remnant*.

The *Product* is the one that remains in the `x>0` region of the machine reference; usually the *Scheme* is arranged to have the *Scraps* as *Remnant*.

When a *Part* contains no further cuts, it is a *leaf* of the tree, and it corresponds to a finished *Piece* or a *Scrap*.

If a *Subsheet* has first-level cuts that are not vertical, a `90°` rotation will be necessary to continue the processing; the rotation direction affects the output sequence of the pieces, given by navigating the tree along the branches on the right (*right-to-left traversal*).

14 *Guillotine* cuts that completely traverse the region to which they are applied.

We can summarize the cuts sequence criteria:

- Consider the current main cut level
  - If its cuts are vertical (parallel to the cutting axis) proceed with the first; note that a cut is not necessarily always completed, it may not involve separation (ex. just score)
  - If not (and there are nested levels), proceed to rotate the *Subsheet*; the direction of rotation (`±90°`) depends on the machine (see next section)

7.4.3 *Sheet* rotations

The direction/sign of rotation is usually expressed relative to the *Sheet/Scheme* reference[15]. In "*straight*" machines the *Sheet/Scheme* reference is left-handed, so negative rotations are counterclockwise and positive rotations are clockwise, as shown in the figure.

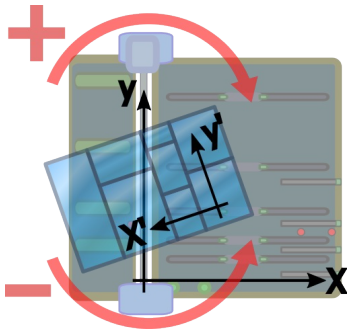


Figure 5: Signs rotations on a straight Strato machine

> On **MACOTEC** machines the *Sheet* rotations are almost always negative.

For generality it's safer to assume that the actual direction of rotations depends on the machine, which declares its preferred rotations on connection (see `rot-dir-prod`, `rot-dir-incoming`), and it is advisable to consider these rather than making assumptions beforehand; however, if forced to make an assumption, a fairly good approximation is to assume that the *Strato* machine always rotate in a negative direction.

> Predicting the direction of rotations is not always essential given the invariance to rotations of `k·180°` for homogeneous rectangles, but it can be important in cases of defective or non-rectangular *Sheets* or Combined processing (see sect. 9 below on p.137).

There are two types of rotations: the rotation of a *Product* just after a cut and the rotation of an incoming *Subsheet* (extracted from the *Feed* table or *Remnant* of a preceding cut).
The first follows the cut of the current *Step*, while the second precedes it.

> *Product* rotations are always preferable because the machine knows already the actual size and position of the glass.

15 The sign of rotation is exactly the opposite in the machine reference.

7.5 *Scheme* generation rules

The main criteria for generating the *Schemes* are minimizing and defragmenting waste, merging the *Scraps* where possible. These first two criteria are already in conflict.

In case of full automatic machines, things get even more complicated because other aspects come into play: processing time and manual interventions may cost more than the glass waste, so the *Scheme* should also be arranged to maximize the automatic operation.

These new requirements typically conflict with the others, and it's up to the final customer to choose the preferred trade-off between them.

Below we discuss the optimization criteria that allow automatic machines to work at their best, minimizing processing times and the manual interventions required from the operator.

7.5.1 Criteria for minimizing processing time

1) Transportability: avoid having to insert or extract *Subsheets* from the *Feed* or *Buffer* that are too small to be safely transported by the belts. (`shift-width-min`, `shift-height-min`).
   - Keep narrow stripes forward to minimize transport, as they will already be in the processing zone
2) Alignability: the longitudinal dimension (width) of the *Product* must be less than the maximum alignable size (`align-max`)
   - Possibility to align the *Remnant*, swapping it with the *Product* (*cut by difference*)
3) Scorability: the transverse dimension (height) of the *Subsheet* must be less than the maximum allowed dimension (`cut-length-max`)
   - Cut the smaller dimension first to shorten the longer one
4) Openability: the width of *Product* and *Remnant* must be such that they can be appropriately handled (clamp-bar distance, breakout counter-wheel width).
   - Condition generally met by other constraints
5) Separability: the width of *Product* and *Remnant* must be such that they do not fall into the *cut area* and can be clamped (`prod-dtch-width-min`, `remn-dtch-width-min`)
   - Arrange the narrow pieces ad *Remnant* on the *Detach* side (the fall conditions of the *Product* are less favorable)
   - Group the narrow pieces and just score without separation, pushing with the *Align* blocks (*sequential cuts*).
6) Minimize the accumulation of *Subsheets* in *Feed*: if full the automatic workflow is blocked
   - Keep the immediate cuts forward and shift the nested cuts towards the end of the *Stripe*, to minimize the insertions into *Feed*
7) Minimize rotations: they are costly operations
   - Nest cuts only when absolutely necessary, and prioritize consecutive cuts on the same level
8) Minimize the rotations of incoming *Subsheets*: they are more expensive compared to those of the *Product* because they require additional alignment and search operations
   - Don’t penalize too much the *Scrap* left at the tail of *Subsheets*
9) Avoid unnecessary initial rotations
   - The first-level cuts of the *Scheme* must be vertical, parallel to the *cutting axis*
10) Rotatability: the *Subsheet* must be compatible with automatic rotation when required
    - Dimensions must not exceed the designated rotation area (threshold for width, height, diagonal).
    - Can be grabbed: the height must be greater than a certain threshold

7.5.2 Exemplification of criteria

Minimizing the accumulation of Subsheets in Feed

To avoid unnecessarily filling the *Feed* table with accumulated *Subsheets*, it's advisable to:

- Move the nested cuts towards the end[16] of the *Subsheet*

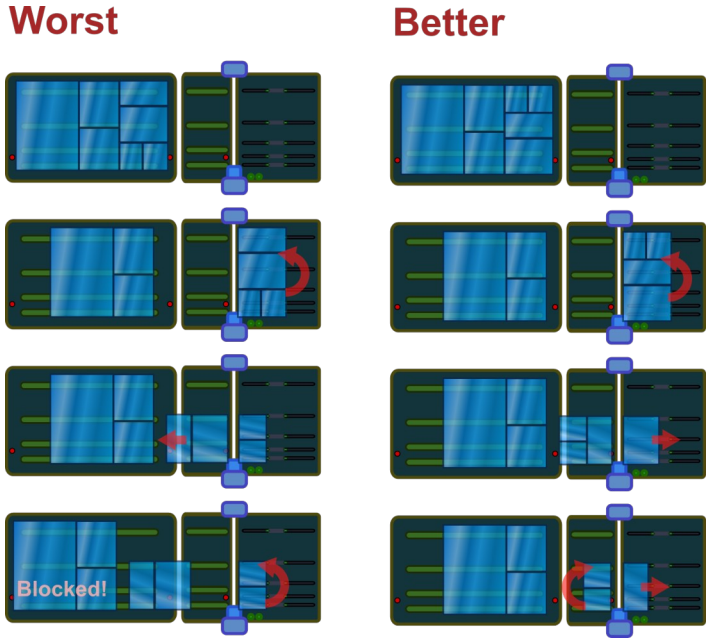
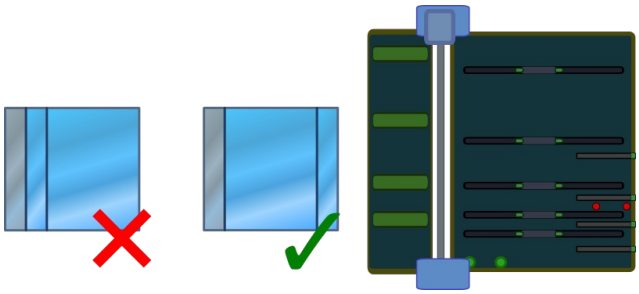


Figure 6: Sequence: minimizing insertions in Feed

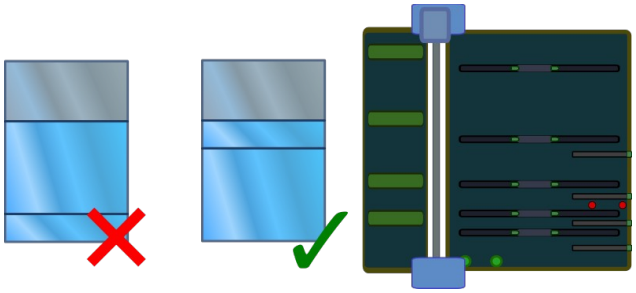
Maximize automatic separations

Avoid ending up with a glass too small to be automatically separated: in such cases, it's better to start cutting the small pieces first to always have enough glass for automatic handling (see `fall-width`):



Minimize movements after Product rotation

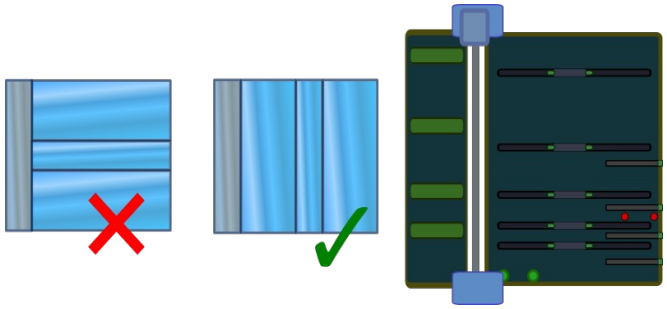
If all cuts are separable, prioritize the larger cuts first to avoid moving the glass back after rotation (and to improve the space needed for any possible subsequent rotations):



16 The term *end* makes sense in relation to the sequencing criteria introduced in 7.4.2 “Cuts sequence and cuts tree” above on p.62

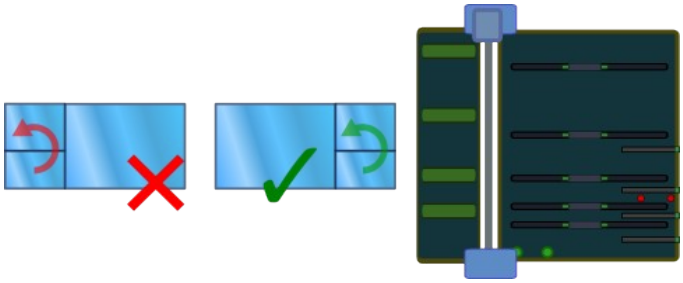
Avoiding unnecessary rotations

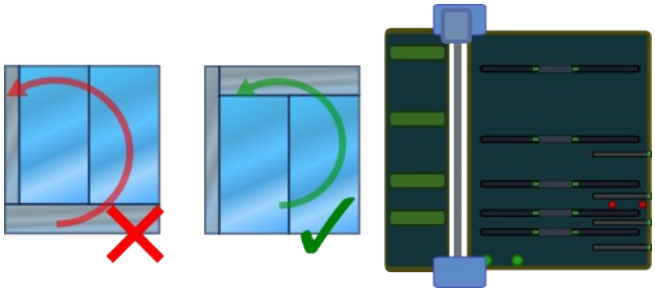
Rotating the glass is a costly operation: it is always advisable to avoid it whenever possible:



Avoiding incoming Subsheets rotations

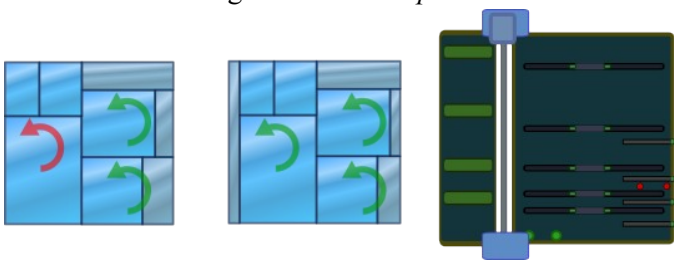
Rotations of the incoming *Subsheets* are particularly costly and should be avoided whenever possible. Prioritize first-level cuts and bring the nested ones forward:





Avoiding non-feasible incoming Subsheet rotations

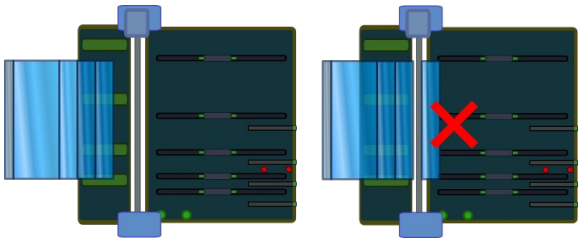
In the event that the incoming *Subsheet* cannot be automatically rotated, in some cases it is possible to transform it into a *Product* rotation by forcing a tail *trim*. However, this measure is not always advantageous: it adds an extra cut and fragments the S*crap*:



⚠Note that these expedients are not always cost-free: in some cases, they may fragment the waste.

7.5.3 Automatic separation of narrow pieces

Without any precaution, it is not possible to automatically separate pieces narrower than `200 mm` on the *Align* side because they would fall into the *cut area*[17]. Let's consider the *Subsheet* below: we usually start with the foremost cut: the *Product* is the finished piece, the *Remnant* contains the subsequent cuts and must move forward to continue the work.



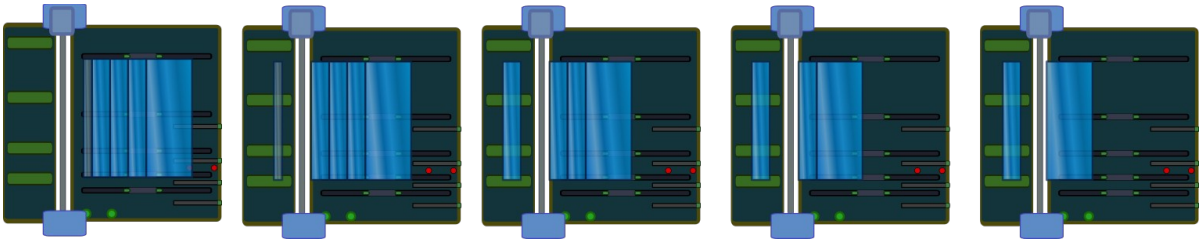
A first possibility is not to request a separation right away (`no-detach`): this function is called "*sequential scores*", the operator must eventually finish the separation by hand after all the cuts has been scored.

On the other hand, if full separation is requested (`full-cut`), the machine will stop at each cut requiring the operator to complete the separation by hand right away.

A first expedient for automatically separating narrow pieces on the *Align* side is to hold them with the clamp and hook them with the align blocks before they fall: here the constraint is that they must be clampable, typically we arrive at minimum widths[18] of `110÷120 mm`.

A second expedient is to have the *Pieces* on the *Detach* side where we can separate automatically *Trims* down to 20mm wide[19].

Considering the example in the figure above, if we rearrange the *Scheme* to cut backward, the machine could complete the separation automatically thanks to this *Trim* trick:



Of course the machine will require the operator to pick up the *Pieces* left on the *Detach*. This method of working is unusual: we cut the *Subsheet* backward, the *Remnant* is the finished *Piece* while the *Product* contains the subsequent cuts and therefore is gradually pushed backward by the *Align* blocks. The wider pieces are placed forward to have the bigger glass on the *Align* side.

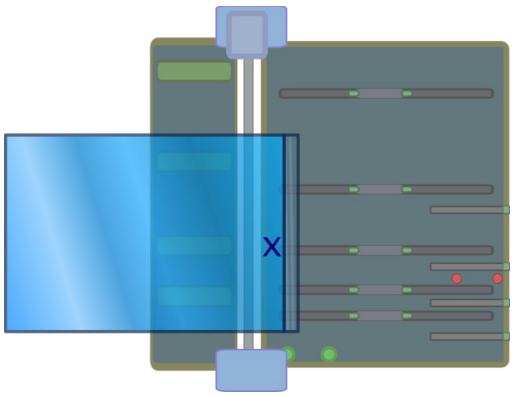
Another solution is to have a *PVB blade* towards the *Align* side (typically mounted on the lower carriage); in this case the scored glass is moved forward before detaching, so that the *Product* is supported as much as possible and cannot fall (head trim sequence).

17 Check `fall-width`
18 Check `prod-dtch-width-min`
19 Check `remn-dtch-width-min`

7.5.4 *Trims* management in Strato processing

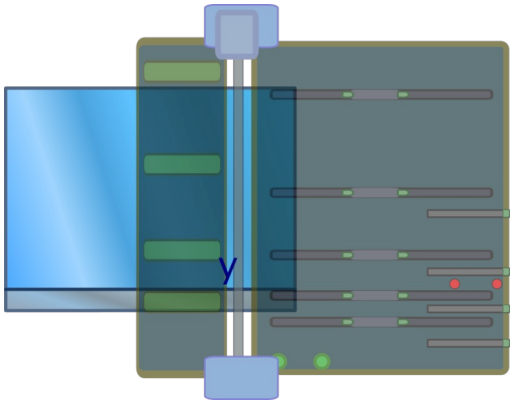
Sheet Trims are an exception in laminated glass cutting.

Any vertical *Trim* is an X cut (also called *head Trim* because placed on the front of the *Sheet*), and therefore does not break the usual assumption that the first-level cuts are vertical and do not involve rotations.



It may only lead to machine automation challenges, such as how to align the cut (if the trim is to be cut, it means the front edge is damaged or not square), and how to automatically separate it, either by providing a specific additional *PVC Blade* or by cutting the *Stripe* first and then rotating it by 180°.

Any horizontal *Trim*, which runs the entire length of the *Sheet*, introduces a description issue: the *Scheme* would start with a Y cut, which requires a rotation before being processed.



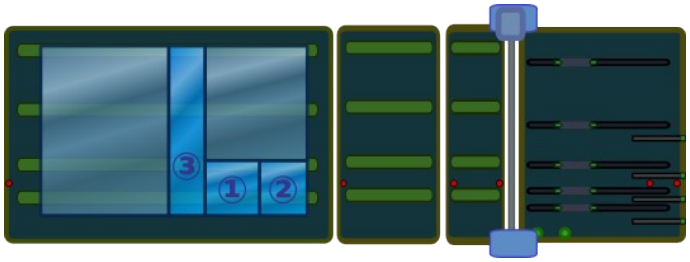
Some implementations may not support this description: it is better to avoid generating *Schemes* with horizontal Y trims, appropriately notifying this limitation.

The user has essentially two solutions:

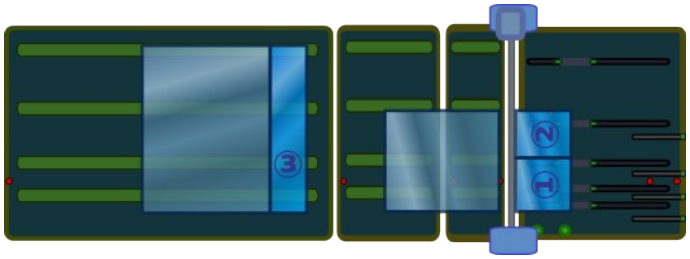
1) Cut the *Trim* manually and re-optimize with the new *Sheet* dimensions
2) Rotate the *Sheet* and re-optimize it with swapped dimensions: the *Trim* becomes vertical
3) Distribute the *Trim* along the *Stripes*, so the Trim becomes a children of the X cuts

7.5.5 Handling big scraps

In case of large *Scraps* it is not always possible to expect the operator to pick them up immediately, for example in this case:

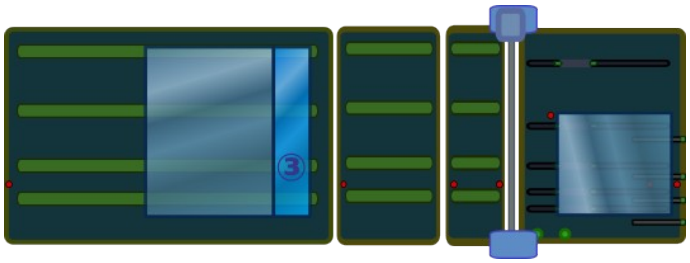


We encounter a rather large *Scrap* on the *Detach*, that is not easily removable by hand:

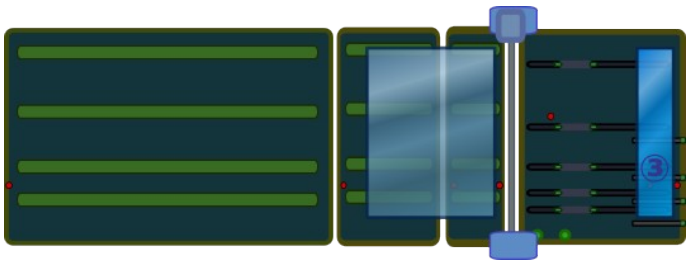


In such cases, it may be convenient provide the option to temporarily insert it into the *Feed* to continue processing:

To then move out it at the *Line-end*, where retrieval is easier. This of course before extracting the *Subsheet* containing the *Piece* 3:



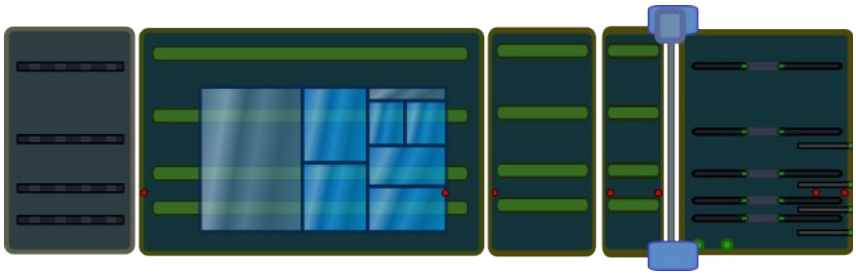
The main *Scrap* of the *Sheet* can also be reinserted in *Feed* or moved out at the *Line-end* depending on the settings:



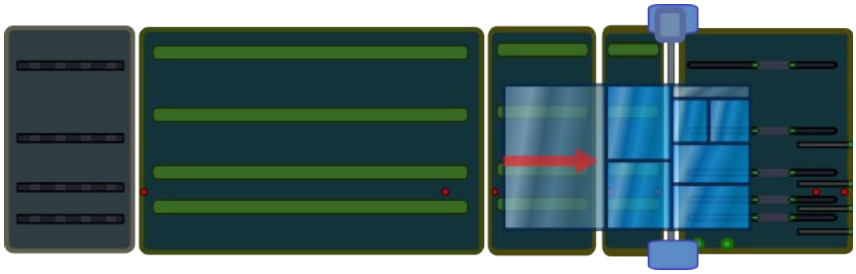
7.5.6 Example of automatic work

Let's review an example of processing on a *Strato* machines equipped with conveyor belts and a rotation device for automatic glass handling.

A *Sheet* is ready on the *Feed* table: after transmitting the job with the *HMI*, the automatic processing can be started pushing the start button. Currently there is no control on the presence of glass in the *cut area*, so it is important for the operator to check the actual state of the machine to avoid dangerous collisions.

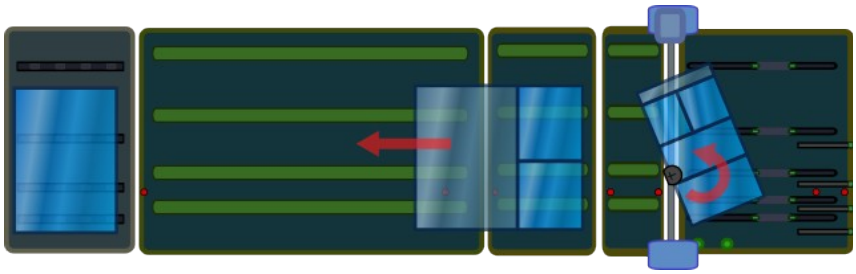


The first cut to be made is already vertical, so the *Sheet* is translated and directly aligned at the cut desired dimension, obtaining the first *Stripe* of the *Scheme*.



Once the separation has been made, the *Remnant* contains other cuts and so is re-inserted[20] into the *Feed* module, to free up the processing zone for the just cut *Stripe*.

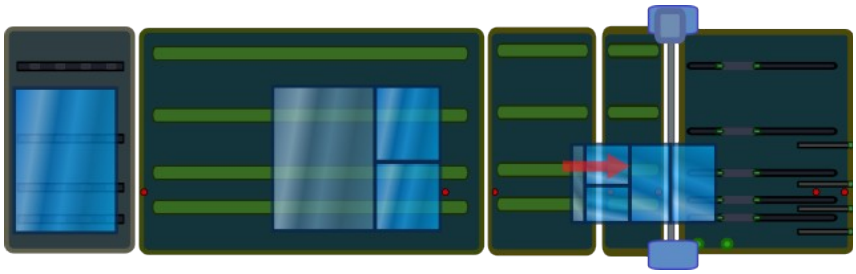
Since the next cut is not vertical, it is necessary to rotate the *Stripe* to proceed: the rotation can start as soon as the necessary area on the *Detach* side is cleared.



The *Remnant* is placed on the *Feed* in order to keep an optimal distance with the subsequent accumulated *Subsheet*, maximizing the number of *Subsheets* that can be accumulated on the *Feed* table.

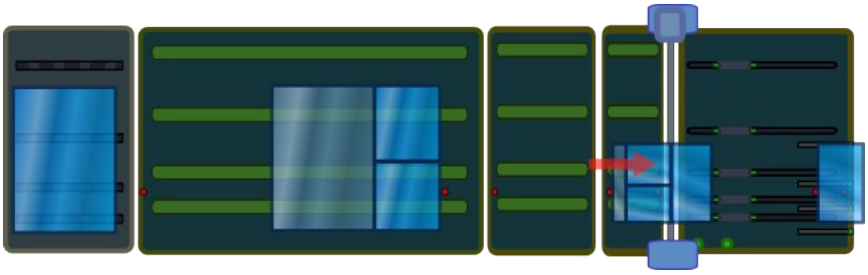
20 Sometimes, in the glass cutting jargon, it's said that it gets “parked”. Given the FIFO nature of the module, it would be appropriate to use the verb “stacked”.

Meanwhile the *Stripe* completes its rotation and is aligned at the position of its first cut.



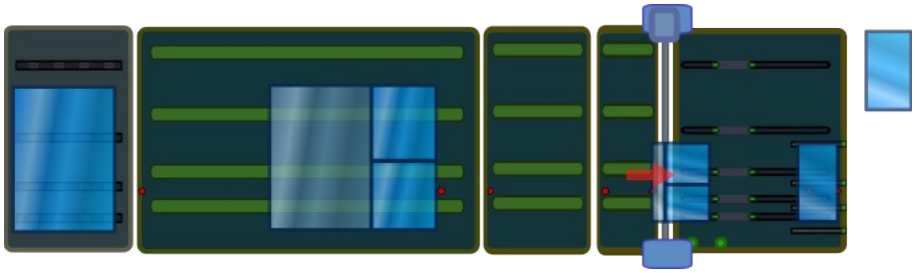
Since the *Product* is a finished *Piece*, is transported to the *Line-end*/*Out-zone* where the operator will retrieve it. The *Remnant* remains in the processing area, since is a *Subsheet* containing other cuts.

If the *Piece* retrieval operations do not interfere with the area needed to align the next cut, the machine continues working; the next cut does not require rotation, so the *Subsheet* is moved forward to be aligned.



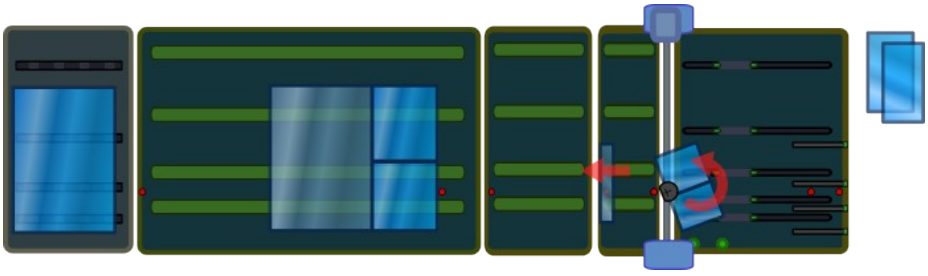
As soon as the *Out-zone* is clear, the next finished *Piece* is moved there.

The next cut is a *Trim*, so the opening and separation sequence will likely be different, but this does not affect the overall workflow sequence, so it is irrelevant whether the *HMI* is aware of it or not.



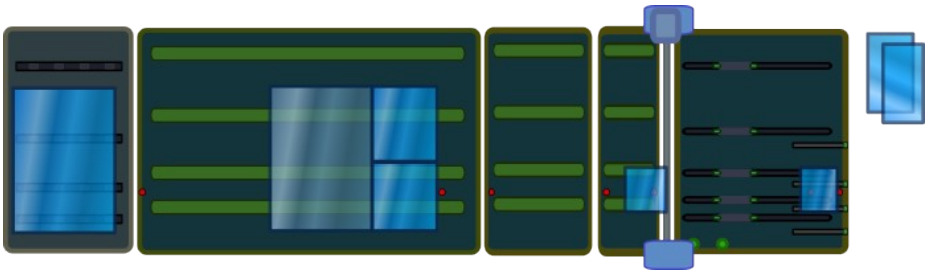
After the separation, we have for the first time a *Remnant* without cuts, a *Trim* that may be automatically dumped in a special bin under the *Detach*. The *Product* contains a cut that is not parallel to the Y-axis: a rotation will be necessary to align it with the *cutting axis*.

As soon as the dumping sequence of the *Trim* finishes, the *Product* will be rotated. Note that the dumping of the *Scrap* may not be done automatically (depending on the *Scrap* size, machine options, absence of a bin, etc.). In such cases the machine will request a manual intervention, and the rotation will have to wait.

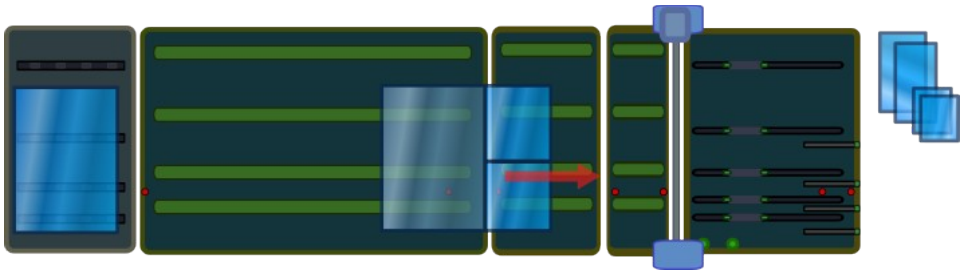


Once rotated, the *Subsheet* is brought forward, aligned and cut. Now both the *Remnant* and the *Product* have no more cuts: typically the *Product* is a finished *Piece* (but could be a *Scrap* sometimes), and the *Remnant* can be either a finished *Piece* or a *Scrap*.

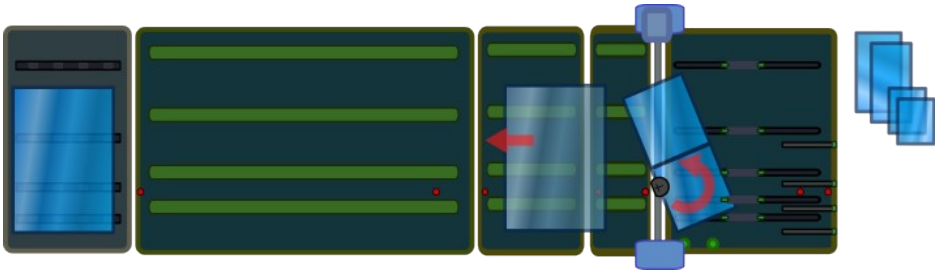
The machine decides how the *Remnant* should be retrieved by the operator: if it is a small *Scrap* it can be dumped automatically, otherwise it can be retrieved by the operator either from the *Detach* side or at *Line-end* (easier in case of big *Scraps*).



Now that the *Subsheet* has been completely cut, it's time to process the *Remnant* previously inserted in *Feed*; it is brought forward and aligned since the cut to be made is already vertical.

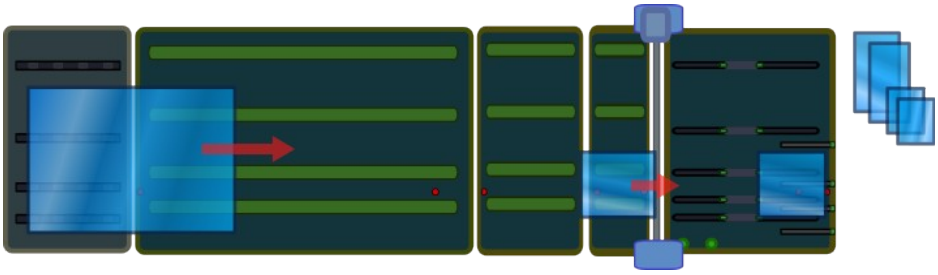


The *Remnant* of this cut constitutes the main *Scrap* of the *Sheet*. Since the next cut involves rotation of the *Product*, the *Remnant* must be either picked up or brought back in order to clear the space needed for the rotation.



Once the rotation is complete, the machine proceeds to perform the last cut.

During the last cut, if the *Feed* table is free, it is possible to proceed to load the next *Sheet* from the *Loader* or by tilting the *Feed* table.

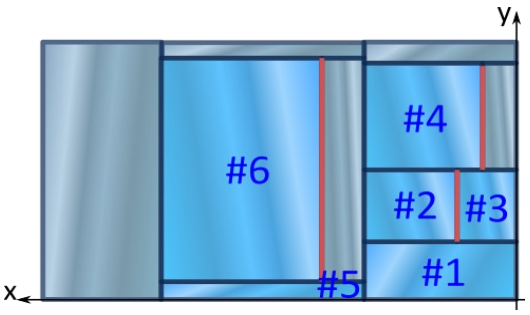


7.5.7 Nested cuts pre-scoring

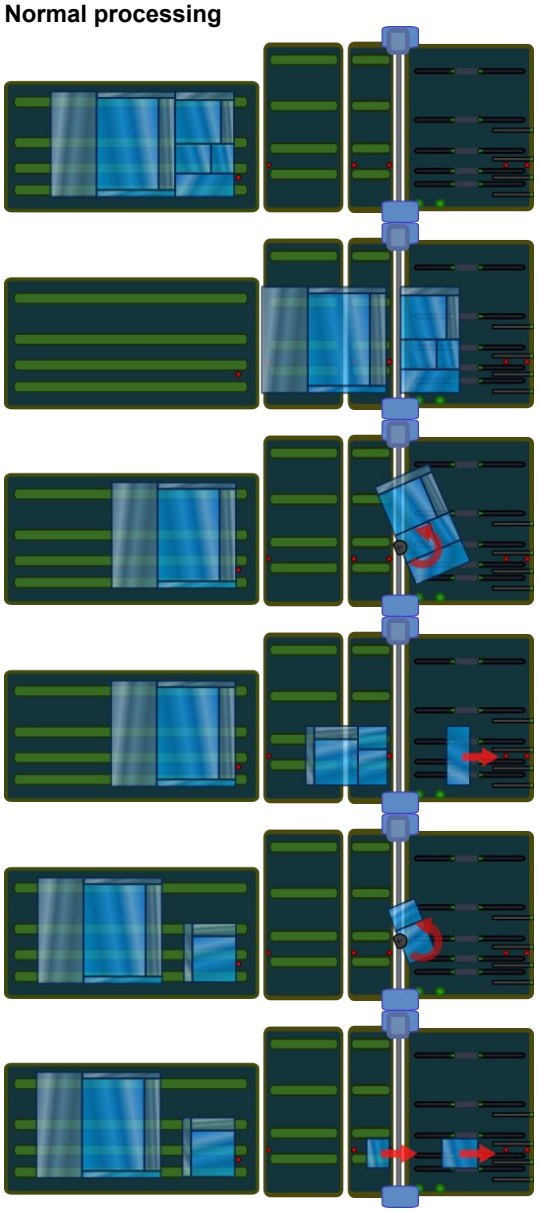
As we've seen in the previous example, processing a nested cut requires rotating the *Subsheet*. However, Z cuts are parallel to X cuts and typically smaller: to reduce overall processing time, we could simply pre-score the Z cuts, thus avoiding the additional rotation needed for their automatic separation.

Let's consider the example in the adjacent figure, we have two *Stripes*, each containing at least one nested Z cut (highlighted in red).

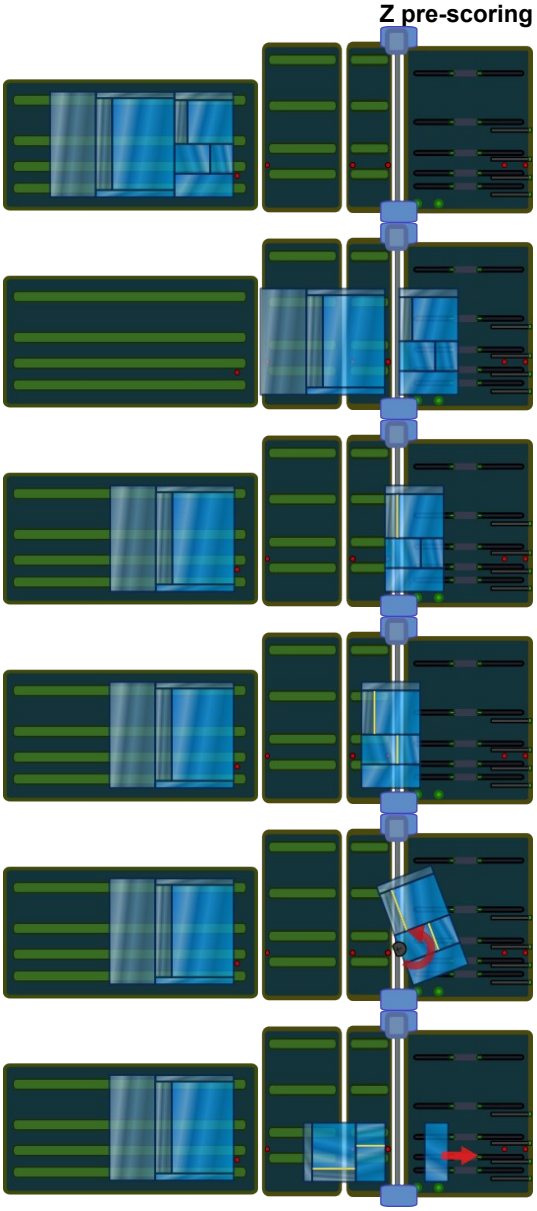
Let's see how it could be processed: for comparison, in the left column, there's the normal processing, while in the right, the nested cuts are pre-scored and then "forgotten": their separation is carried out by the operator, reducing the overall processing time.

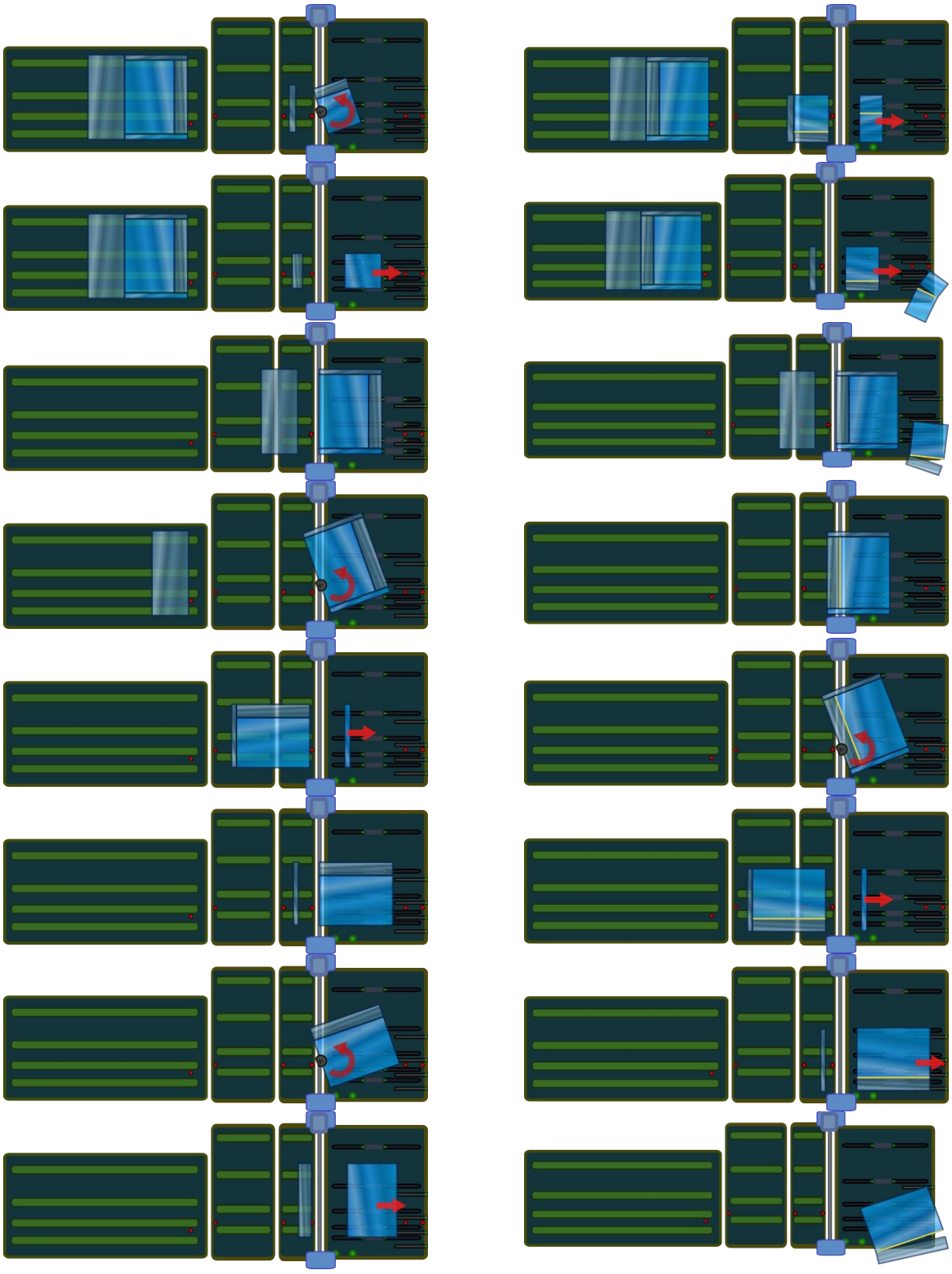


Normal processing



Z pre-scoring





7.6 Strato machine characteristics

On connection (see sect. 3.6.1 “Greet message” above on p.25) the machine transmits the following specific parameters[21] that characterize its features:

`mach-capabilities`	<table><tr><td>`probe`</td><td>Equipped with glass search probe(s)</td></tr><tr><td>`lowe`</td><td>Equipped with low-E wheel</td></tr><tr><td>`cwheel`</td><td>Equipped with counter-wheels for *Trims* breakout</td></tr><tr><td>`label`</td><td>Equipped with a piece labeling system</td></tr><tr><td>`buffer`</td><td>Equipped with a *Buffer* module</td></tr><tr><td>`sep-align-blocks`</td><td>Equipped with independent align blocks</td></tr><tr><td>`shift`</td><td>Able to move *Sheets*</td></tr><tr><td>`rotate`</td><td>Able to rotate *Sheets*</td></tr><tr><td>`zprescore`</td><td>Capable of pre-score Z cuts</td></tr><tr><td>`slantcuts`</td><td>Able to perform oblique cuts</td></tr><tr><td>`shapecurv`</td><td>Capable of machining curvilinear shapes</td></tr></table>
`shift-thickness-max`	*Maximum (overall) thickness automatically movable*
`shift-*-min`	*Minimum size automatically transferable between modules*
`rot-*`	*Size range that can be automatically rotated*
`rot-dir-prod`	*Direction of rotation of the Product after a cut (`-90,+90`)*
`rot-dir-incoming`	*Direction of rotation of an incoming Sheet (`-90,+90`)*
`cut-length-*`	*Cuttable size range*
`line-width-front`	*Overall horizontal dimension of the modules upstream of the cut axis*
`line-width-rear`	*Overall horizontal dimension of the modules downstream of the cut axis*
`line-height`	*Overall vertical dimension of the Line*
`buffer-width`	*Horizontal dimension of the Buffer module*
`align-max`	*Maximum alignable dimension*
`align-height-min`	*Minimum height that can be pushed with the align blocks*
`fall-width`	*Minimum glass width to avoid risk of falling into cut area*
`layer-thickness-max`	*Maximum cuttable single glass layer thickness*
`pvb-thickness-max`	*Maximum cuttable PVB plastic thickness*
`remn-open-width-min`	*Minimum Remnant width for automatic opening*
`remn-dtch-width-min`	*Minimum Remnant width for automatic separation*
`prod-open-width-min`	*Minimum Product width for automatic opening*
`prod-dtch-width-min`	*Minimum Product width for automatic separation*
`slant-*`	*Configurations related to oblique cuts*

21 For details refer to `Interface.xml` *file*

7.7 Material parameters

A summary table of material processing parameters[22]:

	sheet-type	*Material type*	float/strato, lowe, filmsup, filminf (float/strato mutually exclusive)
	h-glass	*Glass thicknesses*	sup[mm]/pvb[mm]/inf[mm]
	 h-glass-sup	*Top layer thickness*	[mm] deprecated
	 h-glass-inf	*Bottom layer thickness*	[mm] deprecated
	 h-plast	*PVB plastic thickness*	[mm] deprecated
	cut-recipe	*Cut sequence recipe*	standard, stressed, hybrid, no-heat, no-lamp, no-blade, max-pull, heat-open
	wheel-angle	*Scoring wheel gradation*	[deg]

	f-lowe	*Pushing force of the low-E coating deletion wheel*	[N]
	v-lowe	*Low-E coating deletion speed*	[m/min]
	f-score-sup	*Pushing force of the upper scoring head*	[N]
	f-score-inf	*Pushing force of the lower scoring head*	[N]
	v-score	*Scoring speed*	[m/min]

	h-brkbar	*Breakout bar opening height*	0:auto 1:low 2:high
	t-brkbar	*Breakout bar opening time*	[s]
	f-clamps-brk	*Clamps push during breakout bar*	[N]

22 For details refer to Interface.xml *file*

	f-brkwhl-sup	*Pushing force of the single breakout wheel*	[N]
	v-brkwhl-sup	*Speed of the single breakout wheel*	[m/min]
	f-brkwhl-sup-trim	*Pushing force of upper breakout wheel for trims*	[N]
	f-brkwhl-inf-trim	*Pushing force of lower breakout wheel for trims*	[N]
	v-brkwhl-trim	*Speed of the breakout wheels for trims*	[m/min]

	f-dtch-pull	*Pull force during separation*	[N]
	v-dtch-pull	*Pull speed during separation*	[m/min]
	f-clamps-max	*Maximum clamps admissible push on the class*	[N]

	t-heat-pre	*Warm-up time before pulling*	[s]
	t-heat-pull	*Heating time limit during pulling*	[s]
	t-heat-add	*Additional heating time for trims before pulling*	[s]
	t-heat-cut	*Cut pre-heating time for stressed glass recipes*	[s]

The *HMI* can open a dialog to edit these parameters as described in sect. 3.15 “Editing material parameters” above on p.33.

7.8 Communicating with *Strato* machines

The communication is carried out as described in sect. 3 “Communication protocol” above on p.18.
Let's look at the specific operations for *Strato* machines.
As always, please refer to the file `Interface.xml` for the meaning and interpretation of the fields.

7.8.1 Monitoring the *Strato* machine status

Regarding how to monitor the machine status, see sect. 3.10 “Monitoring the machine status” above on p.28.
The *Strato*-specific monitored fields, added to the common ones listed in sect. 3.10.4 “Common monitored fields” above on p.30, are:

`mode`	Current working mode
`step`	Index (1..N) of the current *Step*
`proc-done`	Processing/cut done signal
`piece-delivering`	Identifying index of the *Piece* being delivered
`proc-op, cut-recipe`	Work selectors status

For status visualization, the following can also be useful:

`align-pos`	Current position of the align blocks

Significant events

`working=1➞0`
`and proc-done=1`
`and mode=manual`
	Manual cut finished	Move on to the next one?

`proc-done=0➞1`
`and working=1`
`and mode=auto`
	Automatic cut finished	View/print obtained *Piece(s)*
		(see `piece-delivering`)

`step=n➞n+1` `and mode=auto`	Previous (automatic) *Step* completed	Display the next *Step*
`scheme-done=0➞1` `and can-receive-job=1`	*Scheme* finished	Send the next one if present

`can-receive-job=0➞1`
`and scheme-done=1`
`and mode=auto`		

Starting from version `2025-06` of MacoLayer, the `msg="event"` notification is sent to signal a work event. See sect.3.11 “Events notification” above at p.31

7.8.2 Controls and selectors

The *HMI* allows the operator to select and transmit a *Job*, set the speed override, and choose the desired cutting sequence. To get a concrete example, let's see how an old *HMI* appears:

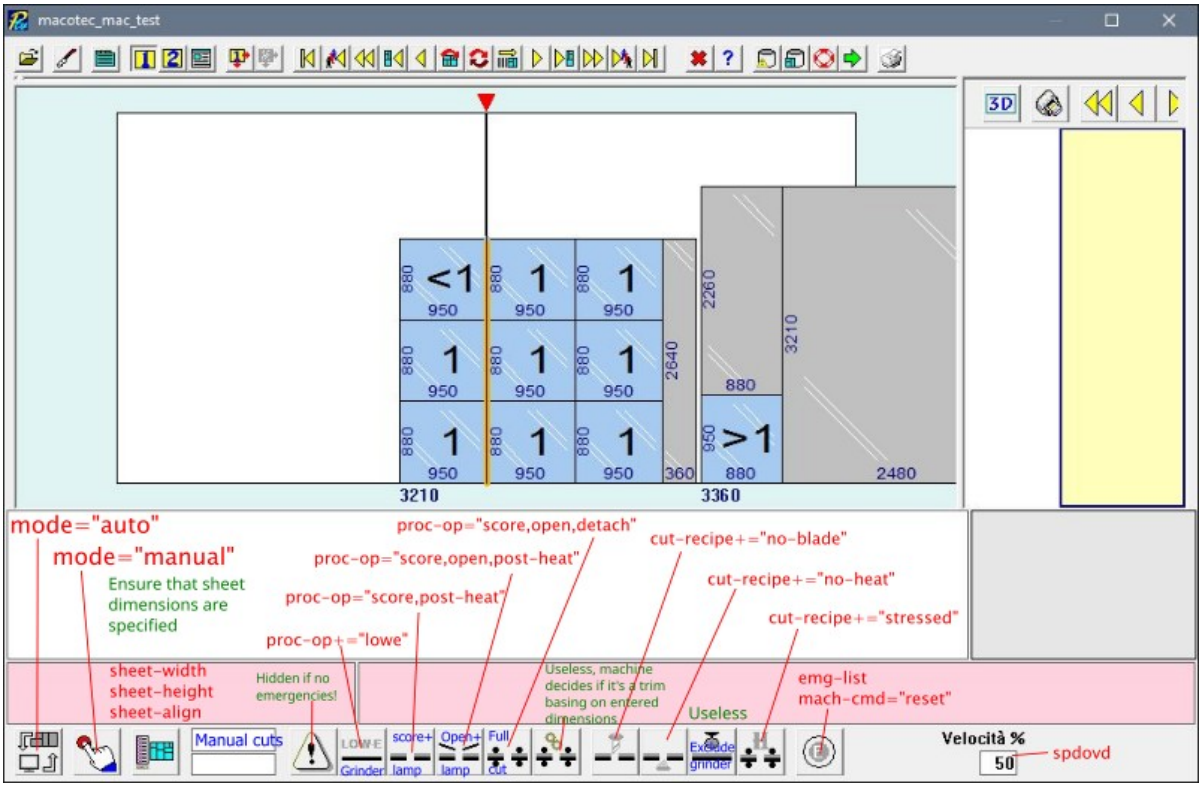


Figure 7: Old HMI for Strato machines

For example on the change event of the speed override control, the *HMI* writes the field:

⇨ `[§]id=1531[§]spdovd=95[ETX]`

⇦ `[§]rep-to=1531[ETX]`

Regarding the selectors that determine how the glass is cut, they are divided into two groups:

- Cut phases: selectors to enable or disable the various phases of the cutting sequence (opening, separation, etc.), reinitialized on selecting a new *Step*
- Cut recipe: select a particular cutting method (special sequence for stressed glass, no lamp, etc.)

When the operator acts on the widgets (on-change event) the *HMI* will send a corresponding write request, for example[23]:

⇨ `[§]id=1532[§]proc-op="open,detach";cut-recipe="no-blade"[ETX]`

⇦ `[§]rep-to=1532[ETX]`

On the next read, the state of the widgets will reflect the actual value of the fields.

> ⚠ The click event NEVER changes the widget state, it only generates a write request. The state will be set on the next reading of the actual value, performed through the status monitoring (see sect. 7.8.1 above).

> ℹ In this teaching example, the operator managed to deselect the scoring and select the separation without the blade in a very short time.

23 For supported values, refer to the file `Interface.xml`

Cut phase selectors

The three selectable values for `proc-op` are: "*score only*", "*score+open*", "*full-cut*". Since these three values are mutually exclusive, for them it's possible to use a single widget having three states (four if we count an unrepresentable state). If the machine is equipped with a low-E wheel, a further selector ("*low-E*") may be useful to allow the operator to skip the lowe wheel pass.

The state of these widgets reflects the type of cut that the machine will perform, reflecting the content of the `proc-op` field read from the machine (see `$status`), which consists of a list of comma-separated strings. Pressing one of these selectors does not change right away the widget's state; rather, it generates a write request containing `proc-op="…"`.

When the `proc-op` field is read by status monitoring (see p.78 above), the widget change its state to reflect its actual value. In non-optimized pseudocode:

```
// New 'proc-op' value read from machine: update selectors status
// Cut phase selectors (mutually exclusive)
if( proc-op.contains("full-cut") or
    (proc-op.contains("score") and proc-op.contains("open") and proc-op.contains("detach")) )
   show_select_just_full_cut();
else if( proc-op.contains("score") and proc-op.contains("open") )
   show_select_just_score_plus_open();
else if( proc-op.contains("score") )
   show_select_just_score_only();
else
   show_nothing_selected();
// Enable low-E (independent)
if( material_is_lowe() and (proc-op.contains("full-cut") or (proc-op.contains("lowe")) )
   show_select_lowe();
else
   show_deselect_lowe();
```

The other event that triggers a `proc-op` write is a manual cut synchronization (see next page); in this case, there are two options: forcing always an initial full cut (`proc-op="full-cut"`), or sending the value corresponding to the current state of the widgets.

Cut recipe selectors

These boolean widgets switch the presence of the respective recipe in the value (comma-separated strings) of the `cut-recipe` field.

Every change generates a write of the field `cut-recipe`: the string related to the selector is added or removed from the current value (last monitored value), for example, if the current value of `cut-recipe` is `"stressed"`, activating the "exclude lamp" selector will generate the write `cut-recipe="stressed,no-lamp"`, and deselecting it `cut-recipe="stressed"`.

As always, the state of the widget s is set on subsequent readings of the field actual value.

7.8.3 Machine visualization

A view of the machine can be created using the following data communicated on connection[24]:

`line-width-front`
`line-width-rear`
`line-height`
	Overall Line dimension before and after the cut axis

`buffer-width`	*Buffer* module size (only if `mach-capabilities` contains `buffer`)
`cut-length-max`	From which can be inferred the cutting *Bridge* length
`align-max`	From which can be inferred the width of the *Align* module

24 See sect. 7.6 "Strato machine characteristics" above on p.75

7.8.4 Synchronizing a manual cut

For example, to cut 1000 mm from a glass sheet 3200x2400 of material 66.1:

```
⇨ [§]id=143[¦]mode="manual"; sheet-size=3200x2400; proc-op="full-cut";
  sheet-align=1000; glass-id="66.1"; sheet-type="strato"; h-glass=6/0.38/6[£]
  ⇦ [§]rep-to=143[£]
```

> In older versions (<=2022) the sheet size were defined with two different fields, now deprecated:
>
> ```
> ⇨ [§]id=143[¦]mode="manual"; sheet-width=3200; sheet-height=2400;
>   sheet-align=1000; proc-op="score,open,detach"; glass-id="66.1"[£]
>   ⇦ [§]rep-to=143[£]
> ```

> When switching between manual and automatic mode you must always specify the material (glass-id), to ensure that you are cutting with the correct material parameters. You may also provide sheet-type and h-glass to allow coherency checks on the material actually pointed.

During a manual cut

When working falls to 0 with proc-done active, it means the cut has been made and the next cut can be selected.

7.8.5 Material selection

When transmitting a *Job*, it is important to always specify the material to be cut: the operator might switch from one *Job* to another in a non-linear way, depending on local priorities.

The *HMI* only needs to indicate the material name (glass-id), the rest is handled by the machine.

> It is optionally possible to enable some consistency checks by adding fields related to the glass type (sheet-type, h-glass), information usually known to the *Optimizer*.

> Note that material synchronization may overwrite the cut-recipe field set in the *HMI* selectors, in this case the material database overrides the last selection.

It will be the responsibility of the machine to manage the parameters listed in sect. 7.7 “Material parameters” above on p.76, requiring the operator to define them in case of unknown/undefined material.

The *HMI* can open a dialog to edit these parameters as described in sect. 3.15 “Editing material parameters” above on p.33.

> In early versions (<2019), the management of material parameters was the responsibility of the *HMI*, which had to pass all of them:
>
> ```
> [§]id=1433[¦]mode="manual"; sheet-width=2000; sheet-height=3000; sheet-align=1000;
> proc-op="lowe,score,open,detach"; glass-id="33.1BE"; sheet-type="strato,lowe"; h-
> glass-sup=3; h-plast=0.38; h-glass-inf=3; cut-recipe="standard"; wheel-angle="155";
> f-score-inf=10; f-score-sup=10; v-score=100.0; f-lowe=10; v-lowe=100.0; h-brkbar=0;
> t-brkbar=0; f-clamps-brk=5; f-brkwhl-sup=20; v-brkwhl-sup=50.0; f-brkwhl-sup-
> trim=20; f-brkwhl-inf-trim=20; v-brkwhl-trim=50.0; f-dtch-pull=40; v-dtch-pull=0.04;
> f-clamps-max=0; t-heat-pre=20.0; t-heat-pull=20.0; t-heat-add=10.0; t-heat-
> cut=10.0[£]
> ```
>
> This modality is deprecated and only supported for backward compatibility.

7.8.6 Generating a *Strato Project*

The considerations mentioned in the sect. 3.12 “Generating a Project” above on p.32 apply.

The `mac` files that describe a *Job* for *Strato* machines, detailed in sect. 7.9 “Strato Scheme descriptor (mac)” below on p.86, usually contain a `[steps]` section, a list of *Steps* that must be done in order to process the *Scheme*.

To generate them is necessary to know the characteristics of the machine; for some general considerations see sect. 7.4 “Schemes for Strato machines” above on p.62; about the *Steps* generation rules see sect. 7.9.2 “Steps generation rules” below on p.88.

> Since the *HMI* needs to provide a graphical representation for each *Step* to show the user, it usually generates the *Steps* itself.
> If the *HMI* delegates the generation of the processing *Steps* omitting the `[steps]` section, it must then retrieve them from the machine in order to create a proper graphical representation for the user.

7.8.7 Project transmission

See sect. 3.13 “Project transmission” above on p.32.

Essentially once the `mac` files have been generated in a certain folder, the *Project* is transmitted/uploaded with the command similar to:

```
⇨ [SOH]id=1432[STX]prj-name="demo1";prj-path="C:\folder"[ETX]
  ⇦ [SOH]rep-to=1432[ETX]
```

7.8.8 *Scheme*/*Step* synchronization

After a *Project* transmission or navigation (browsing the various *Schemes* and *Steps* with the *HMI*), the only remaining action before start working is the so called *synchronization*, which consists selecting the desired *Scheme*/*Step* to be processed.

To do that, the *HMI* sends a request like:

```
⇨ [SOH]id=1433[STX]mode="auto";scheme=1;sheet=2;step=1;glass-id="33.1";
        sheet-type="strato"; h-glass=3/0.38/3; [ETX]
  ⇦ [SOH]rep-to=1433[ETX]
```

> Always specify the material in `glass-id` to ensure the correct parameters. Failing to do so may result in machine damage and material waste.

> The *sheet* field is optional and represents the *Scheme* repetition.

> To enable consistency checks for the material the optional fields `sheet-type` and `h-glass` can be specified.

Regarding the selection of the material see sect. 7.8.5 “Material selection” above on p.81.

If the response is positive, the machine is ready to work the selected *Job* (in jargon, it’s *synchronized* with the *HMI*); if you want to be sure, you can check that the `job-loaded` field is active (monitored in `$status`).

7.8.9 During automatic work

Once a *Job* is *synchronized* on the machine, the operator can start[25] the machining process: the machine will execute the selected *Step* and autonomously proceed to the next one upon completion. The *HMI* tracks the current *Step* index (`step`) to properly display the ongoing operation; optionally, it can use `scheme-progress` and `scheme-remaining-time` to show the completion status of the *Scheme*.

When `step` increments, the *GUI* is updated to reflect the new *Step* in progress.

During the processing of a *Step* (`working==1`), when `proc-done` goes up, the cut is executed and the machine proceeds to the concluding operations of the *Step* (*Product* rotation, *Piece* delivery, …), so the *HMI* may proceed to show the info of the obtained *Piece* (whose index is `piece-delivering`), print its label if required, etc.

> ⚠ `proc-done==1` does not mean the *Step* is completed!
> The *GUI* should always follow the index in `step`.

When `scheme-done` goes up, the current *Scheme* is fully completed. If another *Scheme* is present, it can be synchronized as soon as `can-receive-job` becomes true.

7.8.10 Completion of a *Scheme*

When the machine finishes processing a *Scheme*, it cannot autonomously select the next one, as this depends on information known only to the *HMI*, such as the *Project*'s overall completion status and *Scheme* multiplicity (repetitions), so the *HMI* must be ready to detect the *Scheme* completion.

Detection by status monitoring

The event of *Scheme* completion can be intercepted by normal status monitoring (see sect. 3.10 “Monitoring the machine status” above on p.28).

In addition to the fields `step` and `working`, which let you know whether the last *Step* of the *Scheme* has been executed[26], there is the boolean `scheme-done`, introduced specifically for this purpose: the *Scheme* can be considered completed when this signal is raised.

On *Scheme* completion, the *HMI* proceeds to synchronize the next *Scheme* to process; if there are no more *Schemes*, the *Project* is completed, in this case the *HMI* may ensure to clear any *Job* left on the machine (see sect. 3.17 “Job desynchronization” above on p.35) and ask the operator what to do: proceed with the next queued *Project*, manually open a new one, reset the completion status of the current one, …

From machine request

On *Scheme* completion the machine might, depending on implementation, send a request asking for the next *Scheme* to be processed:

```
⇦ [§#]id=2003[§x]mode,scheme,sheet,step,glass-id[§x]
```

The *HMI* should reply synchronizing the next *Job* (if exists):

```
⇨ [§#]rep-to=2003[§x]mode="auto"; scheme=3; sheet=1; step=1; glass-id="44.1"[§x]
```

Otherwise, for example the *Project* is completed, the *HMI* will reply with an error like:

```
⇨ [§#]rep-to=2003;ret=6;msg="no more schemes!"[§x]
```

> ℹ Note that the machine asks confirmation of the material, which generally may change from one *Scheme* to another.

> ℹ The loading operations of the next *Sheet* can be anticipated during the execution of the last *Steps* of the previous *Scheme*. Since no specific data is needed for these operations, they can be executed before synchronizing the next *Scheme*.

25 Actually, also the *HMI* can. See sect. 3.16.1 “Requesting a start” above on p.34

26 In the early versions, the condition was indicated by `working` falling during the last `step` of the *Scheme*

7.8.11 Clearing a job on the machine

Send a message as described in sect. 3.17 “Job desynchronization” above on p.35.

The effect is to exit the automatic mode and select an invalid (empty) *Step*, thus inhibiting any machine actions if the start button is pressed.

7.8.12 Loss and resumption of synchronization

During the processing of a *Project* the *HMI* must detect if the machine exits automatic mode[27] (`mode` does not contain `auto`).

In such cases, the *HMI* will enter a “*non-synchronized*” state during which it must be ready to receive notifications such as:

⇦ `[␤]msg="ready"[␣]prj-name="project-name";scheme=2;step=8;glass-id[␣]`

That means “*I’m ready to resume automatic processing, so please re-synchronize at that point, including material*”.

If the incoming fields are coherent with the current *Project*, the *HMI* will proceed to synchronize the indicated `scheme`/`step` to return to the synchronized state:

⇨ `[␤]id=2435[␣]mode="auto";scheme=2;step=8;glass-id="33.1";[␗]`

This mechanism avoids unnecessary (and dangerous) manual interventions by the operator to re-synchronize to the correct *Step*.

If there is no *Step* to synchronize (*Scheme* is complete), the machine will send a notification similar to the following:

⇦ `[␤]msg="ready"[␣]prj-name="project-name";scheme=2;scheme-done=1[␣]`

In the opposite (rather unusual) situation, where the *HMI* is in a non-synchronized state and the machine is working:

⇦ `[␤]rep-to=2008[␣]…;can-receive-job=0;…;job-loaded=1;…`
`mode=auto;…;scheme=4;step=2;…;working=1[␗]`

The lesser of the evils is to resume a *synchronized* state, possibly checking if the *Project* name match (reading from the machine `prj-name`).

7.8.13 Label printing requests

The *HMI* must be ready to respond to any requests from the machine as described in sect. 3.14 “Serving prints” above on p.33.

7.8.14 Measurement notifications

Some machines, if equipped with the appropriate hardware, might send notifications containing measured data.

Measurement of the actual glass thickness

The machine might measure and communicate the actual glass thickness with a notification such as[28]:

⇦ `[␤]msg="data"[␣]sheet-thickness=8.382[␗]`

27 This typically happens when the operator decides to cut a *Shape*.

28 Always consult `Interface.xml` for guidance on interpreting the fields.

7.8.15 *Shape* processing

After transmitting and synchronizing a *Project*, it is possible to request the machine to process one or more *Shape*s of the current *Scheme*:

⇨ `[§]id=31[¦]mode="shape";shape="name1,name2";scheme=2;step=11;glass-id="3.1"[¶]`

> In this request, the type of *Shape* (polygonal or generic) is irrelevant: the machine will behave appropriately.

> In a *Shape* processing request, it's optionally possible to declare additional information such as `scheme`, `step` and `glass-id`. These might be used if a *Job* has not already been synchronized.

Let's assume we've transmitted a *Project* and synchronized a *Scheme* containing two *Shape*s named `"shp1"` and `"shp2"`. Upon synchronizing this *Scheme*, the *HMI* must show a button to process all the *Shape*s in the *Scheme*; if this button is pressed, the *HMI* will send the following request to the machine:

⇨ `[§]id=32[¦]mode="shape";shape="all"[¶]`
⇦ `[§]rep-to=32[¶]`
⇦ `[§]rep-to=32;ret=7;msg="multiple shapes not yet supported"[¶]`

Let's assume the user selects a *Step* which obtains the *Primitive* of `"shp1"`: the *HMI* must show also a button to process this single *Shape*; if this button is pressed, the *HMI* will send the following request to the machine:

⇨ `[§]id=33[¦]mode="shape";shape="shp1"[¶]`
⇦ `[§]rep-to=33[¶]`
⇦ `[§]rep-to=33;ret=3;msg="shape not found"[¶]`

After sending these requests, the machine will likely switch to `mode="manual,shape"` or `mode="manual,slant"`.

The *HMI* must be ready to re-synchronize the *Job* upon receiving the notification that machine is ready to resume automatic mode (see sect. 7.8.12 above):

⇦ `[§]msg="ready"[¦]prj-name="test";scheme=3;step=6;glass-id[¶]`

7.9 *Strato Scheme* descriptor (mac)

The general format described in 4 “Scheme descriptor format (mac)” above on p.36 is specialized for *Strato* machines through the [steps] section, where are listed the *Steps* that the machine must execute to process the *Scheme*.

Building the *Steps* sequence requires considering the specific machine's characteristics to ensure maximum automation and execution speed (maximum alignable dimension, minimum transportable width, preferred rotations, etc.); for further details refer to sect. 7.4 “Schemes for Strato machines” above on p.62.

7.9.1 [steps] section

Instructs the machine on how process a *Scheme*.

Each *Step* is described by the following information, divided into five groups:

<num>="descr", status:<initial status>, sheet:<incoming Sheet info>, op:<processing flags>, prod:<Product info>, remn:<Remnant info>

- status (initial status, expected situation)
  - stack:<int> (number of stacked *Subsheets* in *Feed*)
  - proc:<int> (number of *Subsheets* in processing zone)
- sheet (about the *Subsheet* to process)
  - size:<double>x<double> (dimensions *width*×*height*)
  - extract|pop, out-fwd, out-bck (Glass handling *flags*)
  - rotate:<string> (+90, -90 – rotation before the cut/processing)
  - align:<double> (dimension to align for the cut/processing)
- op (about cut/processing)
  - full-cut, lowe, score, open, detach, no-detach, no-lowe, pre-heat, post-heat (processing *flags*)
    - lowe – include the low-E coating deletion
    - score/open/detach – include the scoring/breakout/separation phase
    - full-cut ≡ (lowe)?, score, open, detach
    - no-detach – exclude detach; if alone, it means full-cut except detach
    - no-lowe – exclude lowe; if alone, it means full-cut except lowe
    - pre-heat – include heat before cut
    - post-heat – include heat after the (not separated) cut
  - y:<double>|<double> or y:(<double>,<double>) (cut/processing ordinates)
- prod (about the *Product* of the cut/processing)
  - size:<double>x<double> (dimensions *width*×*height*)
  - rotate:<string> (rotation after the cut)
  - piece:<int>|<int>|…, scrap (*Piece* indexes/*Scrap* flag)
  - shape:<string> (data of associated *Shape*)
  - out-fwd (Glass handling *flags*)
- remn (about the *Remnant* of the cut/processing)
  - size:<double>x<double> (dimensions *width*×*height*)
  - piece:<int>|<int>|…, scrap (*Piece* indexes/*Scrap* flag)
  - shape:<string> (data of associated *Shape*)
  - insert|push, out-fwd (Glass handling *flags*)

The order of these blocks constituting a *Step* reflects the chronological order of operations and should be respected for readability.

The prod and remn groups are usually specified only when there’s separation; an exception is when it is necessary to declare the type (piece, scrap) or the rotation (rotate), such as in a piece obtained in a low-E deletion-only *Step*.

A few annotated examples may help clarify the concepts:

```
1="X 1000", status:{stack:1 proc:0}, sheet:{size:6000x3210 extract align:1000}, op:{full-cut},
  prod:{size:1000x3210 rotate:-90°}, remn:{size:5000x3210 insert}
```

We have a `6000x3210` *Subsheet* on *Feed* that must be extracted, aligned to `1000`, and cut. The resulting *Product* will be rotated (to be probably processed in the next *Step*), while the *Remnant* will be inserted back in *Feed*.

```
2="Y 1200", status:{stack:1 proc:1}, sheet:{size:3210x1000 align:1200}, op:{full-cut},
  prod:{size:1200x1000 piece:1}, remn:{size:2010x1000 scrap}
```

We have a `3210x1000` *Subsheet* in processing zone that must be aligned to `1200` and cut; the *Product* is *Piece* `1` and the *Remnant* is a *Scrap*: both will be taken by the operator, so this *Step* frees the processing zone: the next *Step* will probably have to extract a *Subsheet* from the *Feed*.

```
4="Y 500", status:{stack:1 proc:1}, sheet:{size:1000x2400 rotate:-90° align:500}, op:{full-cut},
  prod:{size:500x1000 piece:0|2}, remn:{size:1900x1000}
```

We have a `1000x2400` *Subsheet* in processing zone that must be rotated, then aligned to `500` and cut; the *Product* is an aggregate that contains the *Piece* `2`, which will be manually separated and then retrieved, while the *Remnant* remains in processing zone and will probably be processed in the next *Step*.

```
3="Y 0", status:{stack:1 proc:1}, sheet:{size:3210x1948 align:0}, op:{lowe}
```

We have a `3210x1948` *Subsheet* in processing zone that must be aligned to `0` and then processed with the low-E wheel: this is a description of a “head deletion”, where the low-E is deleted from the front side of the *Subsheet*. In this *Step* there’s no glass separation, so there’s no *Remnant*. The *Product* coincides with the original *Subsheet*, which remains in processing zone and will be handled in the next *Step*.

```
2="Z 850 (pre)", status:{stack:1 proc:1}, sheet:{size:1000x2400 align:850}, op:{score y:500|1600}
```

We have a `1000x2400` *Subsheet* in processing zone that must be aligned to `850` and scored from `500` to `1600`. This is a Z-cut pre-scoring.

```
5="Y 1100", status:{stack:1, proc:1}, sheet:{size:1900x1000, align:1100}, op:{full-cut},
   prod:{size:1100x1000, piece:0|4}, remn:{size:800x1000}
```

We have a `1900x1000` *Subsheet* in processing zone that must be aligned to `1100` and cut. The *Product* is an aggregate containing *Piece* 1 (was probably pre-scored in a previous *Step*) and will be delivered to operator, the *Remnant* remains in the process zone for further processing.

```
3="X 1600", status:{stack:0 proc:1}, sheet:{size:3000x3210 align:1600}, op:{open,detach},
  prod:{size:1600x3210 out-fwd}, remn:{size:1400x3210 out-fwd}
```

We have a `3000x3210` *Subsheet* in processing zone whose pre-scored cut must be aligned to `1600` to be opened and separated, finalizing a previously scored cut. Both the *Product* and the *Remnant* will be moved out to *Line-end*.

```
1="OUT", status:{stack:1, proc:0}, sheet:{size:6000x3210, extract, out-fwd}
```

We have a `6000x3210` *Sheet* in *Feed* that needs to be moved out to *Line-end*. It's probably for a combined machine equipped with a breakout table downstream the cut module.

> Using commas is just a matter of style: it's required only when extending a definition over multiple lines, see sect. 11.2 “Extended json-ini syntax” below on p.154.

7.9.2 Steps generation rules

1) *Steps* are numbered starting from `1`
2) The initial state of the first *Step* is always one *Sheet* in *Feed* and nothing in processing zone: `status:{stack:1, proc:0}`;
3) Regarding the *Step* initial state:
   - Both `stack` and `proc` can never be negative; `proc` cannot be greater than `1`
   - If `sheet` contains `pop` (`extract`) then must be `stack>0`
   - If `remn` contains `push` (`insert`) then the next *Step* will have `stack` incremented by `1`
   - If `proc>0` then `sheet` must not contain `pop (extract)`
4) The dimensions of the incoming `sheet` are those before any possible rotations
5) The feasibility of the cut is handled by the machine. If a cut must be separated, the separation flag `op:{detach,…,full-cut}` must be always specified; if it will be executed automatically or manually it's not a concern
6) The notation `op:{full-cut}` implies also the low-E wheel pass if necessary
7) If there is not separation `remn` should never be specified; `prod` has the same size of `sheet` and should be specified only when it's necessary to indicate its attributes (`piece`, `scrap`, `shape`, `out-fwd`)
8) If there is not separation `prod` shouldn't contain `rotate` since any subsequent rotation is done on the `sheet` of the next *Step*; an exception may be when the *Product* is already in position to be rotated (ex. low-E deletion on the backward edge)
9) `remn` never contains `rotate` since any subsequent rotation is done on the `sheet` of the next *Step*
10) When there's separation, `align` coincides with the *Product* width
11) When there's separation and the *Product* needs to be further processed, the `sheet` of the next *Step* will have the same size (swapped if rotated); the *Remnant* must leave the processing zone, so `remn` will contain either `push/insert` or `piece` or `scrap`
12) If a *Product* or *Remnant* contains further processing (ex. even just a low-E wheel pass) it can never be marked as `piece` or `scrap`
13) If `prod` or `remn` is a finished *Piece* (`piece`), it never contains `rotate`: any reorientation to facilitate delivering/retrieval of the *Piece* is handled by the machine
14) If `prod` or `remn` is not a finished *Piece* (`piece`), there is no point in declaring the `shape` field

Note that in the definition of a *Step* contains redundant data, for example `stack`/`proc` and `size` of `sheet`,`prod`,`remn` can often be derived from context, and in that case they could be omitted. These redundancies are intentional and designed to enhance error checking and improve readability.

When the *HMI* specifies the `[steps]` section, the machine is not allowed to alter the declared *Steps*, as doing so could result in the *HMI* visualization becoming misaligned with the machine's actual operations.

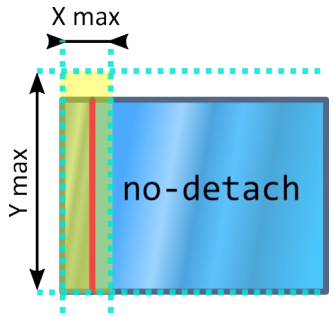
7.9.3 Automatic separation of *Trims*

The sequence to automatically separate *Trims* can be a lengthy and laborious task for the machine. Because of this, operators often prefer to just score them and then separate manually, while the machine continues with its subsequent work.

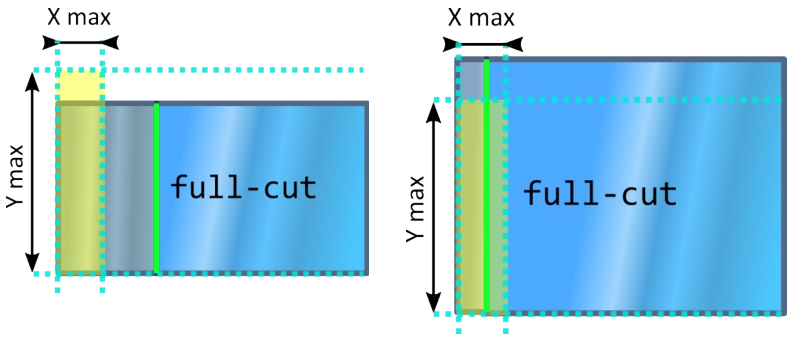
This is why it's helpful to provide dimensional thresholds to indicate when a *Trim* should be just scored and not separated, delivering to the operator the *Piece* still attached to its *Trim*.

These configurable thresholds are useful because the larger the glass, the more burdensome the manual intervention, and each operator has their personal preferences.

Example of *Trim* within the thresholds that won't be separated by the machine:



Example of *Trims* outside thresholds that will be separated automatically by the machine:



This setting will affect the *Step* sequence, for example a *Trim* separated by the machine:

```
7="sep", sheet:{size:800x500, align:700}, op:{full-cut}, prod:{size:700x500, piece:1}, remn:{size:100x500, scrap}
```

Without automatic separation the *Step* becomes:

```
7="just scored", sheet:{size:800x500, align:700}, op:{score}, prod:{size:800x500, piece:1|0}
```

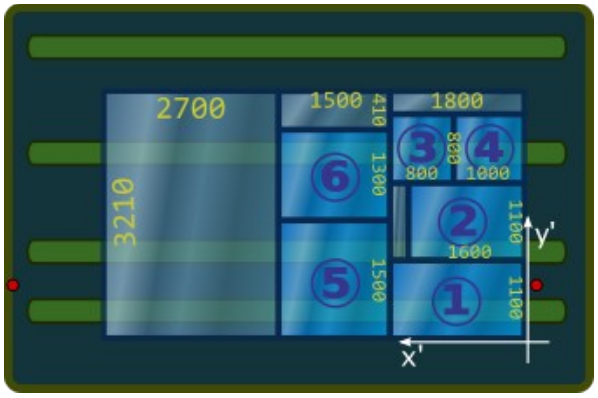
7.9.4 Basic commented example

```
; About the overall order
[project]
name="strato-first-demo"
scheme=1/1 ; Index of scheme/total schemes count


; Directives to importer
[options]
; Declaration of the adopted units of measurement, expressed in the International
System of Units (S.I.)
units =
   {
    length: 0.001 ; [m] Length expressed in millimeters
   }

; Declaration of rounding/tolerances expressed in the chosen units of measurement
granularity =
   {
    length: 0.01 ; [length] Rounding to the nearest hundredth of the specified unit
   }


; About this Scheme
[scheme]
type="strato" ; strato, float, stripes, labels, no-lowe, ...
size=6000x3210 ; [length] Sheet dimensions width x height
glass-id="33.1" ; Material identifier
sheets=2 ; Repetitions/multiplicity
X=1000          ;                    6000
  Y=1200 id:1   ;     +-----+---------+------+---------+
  Y=1500        ;     |--+--|---------|      |         |
    Z=500 id:2  ;     |  |  | #5   950|  //  |         |
    Z=500 id:3  ; 1500|#2|#3|---------|      |    //   | 3210
X=2000          ;     |--+--| #4      |------|         |
  Y=1700 id:4   ; 1200|     |     1700| #6   |         |
  Y=950 id:5    ;     | #1  |         |  1100|         |
X=1100          ;     +-----+---------+------+---------+ -> X
  Y=1100 id:6   ;      1000    2000     1100    1900


; About the obtained Pieces
[pieces]
1={ size:1000x1200 descr:"first" }
2={ size:500x1500  descr:"second" }
3={ size:500x1500  descr:"third" label:0 }
4={ size:2000x1700 descr:"fourth" }
5={ size:2000x950  descr:"fifth" }
6={ size:1100x1100 descr:"sixth" }


; Pieces labeling data ; +-----------+
[labels]               ; | TL  T  TR |
label-size=100x80      ; | L   C   R |
alignment=C            ; | BL  B  BR |
                       ; +-----------+
; Note: Indexes match the corresponding Piece index
1={ pos:(500,600) print:"C:\Labels\Label1.xml" }
2={ pos:(250,1950) print:"C:\Labels\Label2.xml" }
3={ pos:(750,1950) piece:0 }
4={ pos:(2000,850) print:"C:\Labels\Label4.xml" }
5={ pos:(2000,2175) print:"C:\Labels\Label5.xml" }
6={ pos:(3550,550) print:"C:\Labels\Label6.xml" }
```

```
; Sequence of processing steps
[steps]
;idx="descr", status:<initial status>, sheet:<incoming Sheet info>, op:<processing flags>,
     prod:<Product info>, remn:<Remnant info>
; Cut of the first Stripe
1="X 1000", status:{stack:1 proc:0}, sheet:{size:6000x3210 pop align:1000}, op:{full-cut},
  prod:{size:1000x3210 rotate:-90°}, remn:{size:5000x3210 push}
; Processing the first Stripe
2="Y 1200", status:{stack:1 proc:1}, sheet:{size:3210x1000 align:1200}, op:{full-cut},
  prod:{size:1200x1000 piece:1}, remn:{size:2010x1000}
3="Y 900", status:{stack:1 proc:1}, sheet:{size:2010x1000 align:1500}, op:{full-cut},
  prod:{size:1500x1000 rotate:-90°}, remn:{size:510x1000 scrap}
4="Z 500", status:{stack:1 proc:1}, sheet:{size:1000x1500 align:500}, op:{full-cut},
  prod:{size:500x1500 piece:2}, remn:{size:500x1500 piece:3}
; Cut of the second Stripe
5="X 2000", status:{stack:1 proc:0}, sheet:{size:5000x3210 pop align:2000}, op:{full-cut},
  prod:{size:2000x3210 rotate:-90°}, remn:{size:3000x3210 push}
; Processing the second Stripe
6="Y 1700", status:{stack:1 proc:1}, sheet:{size:3210x2000 align:1700}, op:{full-cut},
  prod:{size:1700x2000 piece:4}, remn:{size:1510x2000}
7="Y 950", status:{stack:1 proc:1}, sheet:{size:1510x2000 align:950}, op:{full-cut},
  prod:{size:950x2000 piece:5}, remn:{size:560x2000 scrap}
; Cut of the third Stripe
8="X 1100", status:{stack:1 proc:0}, sheet:{size:3000x3210 pop align:1100}, op:{full-cut},
  prod:{size:1100x3210 rotate:-90°}, remn:{size:1900x3210 scrap push}
; Processing the third Stripe
9="Y 1100", status:{stack:1 proc:1}, sheet:{size:3210x1100 align:1100}, op:{full-cut},
  prod:{size:1100x1100 piece:6}, remn:{size:2110x1100 scrap}
;10="Moving out Sheet main scrap", status:{stack:1 proc:0}, sheet:{size:1900x3210, out-fwd}

; full-cut is an alias for lowe,score,open,detach

; 'stack'/'proc' and 'size' are generally deducible from other data
; These redundancies are intentional, designed to enhance
; error checking and improve human readability

; Other examples

; Head deletion: low-E wheel pass
;1="X 0", status:{stack:1 proc:0}, sheet:{size:6000x3210 pop align:0}, op:{lowe}

; Downstream transfer of the Product
;#="X 1000", status:{stack:0 proc:1}, sheet:{size:6000x3210 pop align:1000}, op:{full-cut},
   prod:{size:1000x3210 out-fwd}, remn:{size:5000x3210}

; Sequential scores
;#="X 10", sheet:{size:2000x1000 align:10}, op:{score,no-detach}
;#="X 10", sheet:{size:2000x1000 align:20}, op:{score,open,no-detach}

; Transferring a Sheet
;#=sheet:{size:2000x1000 pop out-fwd}

; Returning a Sheet to loading system
;#=sheet:{size:2000x1000 out-bck}
```

7.9.5 Annotated example

Let's assume we need to represent the processing of the following *Scheme*:



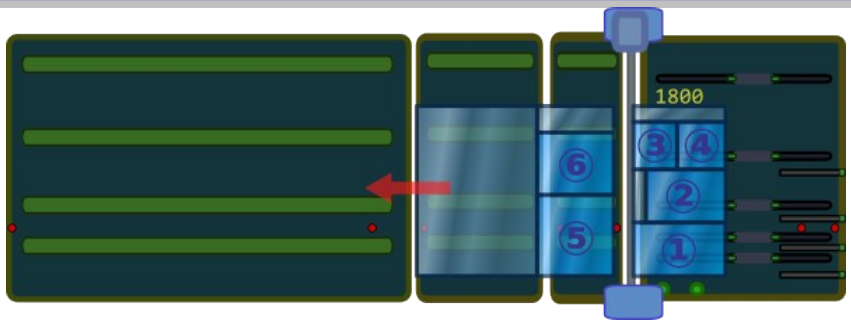
The `mac` file that describes this *Scheme* is:

```
[project]
name="annotated"
scheme=1/1

[scheme]
type="strato"
size=6000x3210
glass-id="66.1" ;       1800         1500          2700
X=1800          ; +------------+----------+----------------+
  Y=1100 id:1   ; +------+-----|    //    |                |
  Y=1100        ; |  #4  | #3  +----------+                |
    Z=1600 id:2 ; +------+---+-+          |                |
  Y=800         ; |          | |    #6    |       //       |
    Z=800 id:3  ; |    #2    | +----------+                |
    Z=1000 id:4 ; +----------+-+          |                |
X=1500          ; |            |    #5    |                |
  Y=1500 id:5   ; |     #1     |          |                |
  Y=1300 id:6   ; +------------+----------+----------------+ --> x

[pieces]
1={ size:1800x1100 }
2={ size:1600x1100 }
3={ size:800x800 }
4={ size:1000x800 }
5={ size:1500x1500 }
6={ size:1500x1300 }

[steps]
1="X 1800", status:{stack:1 proc:0}, sheet:{size:6000x3210 extract align:1800}, op:{full-cut}, prod:{size:1800x3210}, remn:{size:4200x3210 insert}
2="Z 1600 (pre)", status:{stack:1 proc:1}, sheet:{size:1800x3210 align:1600}, op:{score y:1100|2200}
3="Y 1100", status:{stack:1 proc:1}, sheet:{size:1800x3210 rotate:-90 align:1100}, op:{full-cut}, prod:{size:1100x1800 piece:1}, remn:{size:2110x1800}
4="Y 1100", status:{stack:1 proc:1}, sheet:{size:2110x1800 align:1100}, op:{full-cut}, prod:{size:1100x1800 piece:0|2}, remn:{size:1010x1800}
5="Y 800", status:{stack:1 proc:1}, sheet:{size:1010x1800 align:800}, op:{full-cut}, prod:{size:800x1800 rotate:-90}, remn:{size:210x1800 scrap}
6="Z 800", status:{stack:1 proc:1}, sheet:{size:1800x800 align:800}, op:{full-cut}, prod:{size:800x800 piece:3}, remn:{size:1000x800 piece:4}
7="X 1500", status:{stack:1 proc:0}, sheet:{size:4200x3210 extract align:1500}, op:{full-cut}, prod:{size:1500x3210 rotate:-90}, remn:{size:2700x3210 scrap}
8="Y 1500", status:{stack:0 proc:1}, sheet:{size:3210x1500 align:1500}, op:{full-cut}, prod:{size:1500x1500 piece:5}, remn:{size:1710x1500}
9="Y 1300", status:{stack:0 proc:1}, sheet:{size:1710x1500 align:1300}, op:{full-cut}, prod:{size:1300x1500 piece:6}, remn:{size:410x1500 scrap}
```

Let's consider each individual *Step* by visualizing the respective processing.

***Step* 1**

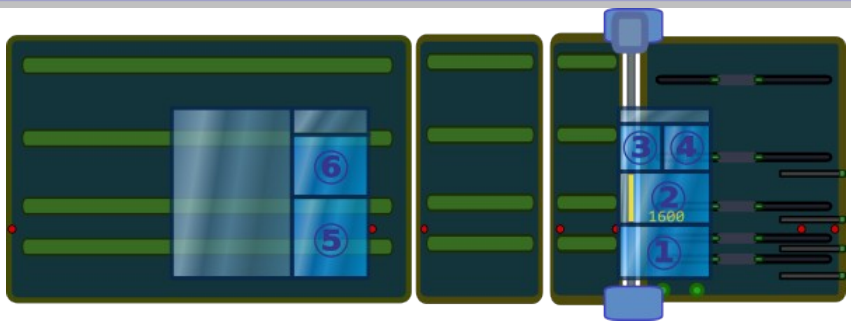
```
1="X 1800", status:{stack:1 proc:0}, sheet:{size:6000x3210 extract align:1800}, op:{full-cut},
  prod:{size:1800x3210}, remn:{size:4200x3210 insert}
```



This is the cut of the first *Stripe*: the initial situation of the first *Step* is always a *Sheet* on *Feed* and a free processing zone (`stack:1 proc:0`). The *Sheet* must be extracted from the *Feed* (`extract` or `pop`), aligned to `1800` and cut (`full-cut` is equivalent to `score,open,detach`). The *Product* remains in the processing zone, the *Remnant* must leave and be reinserted back into *Feed* (`insert` or `push`).

***Step* 2**

```
2="Z 1600 (pre)", status:{stack:1 proc:1}, sheet:{size:1800x3210 align:1600}, op:{score y:1100|2200}
```

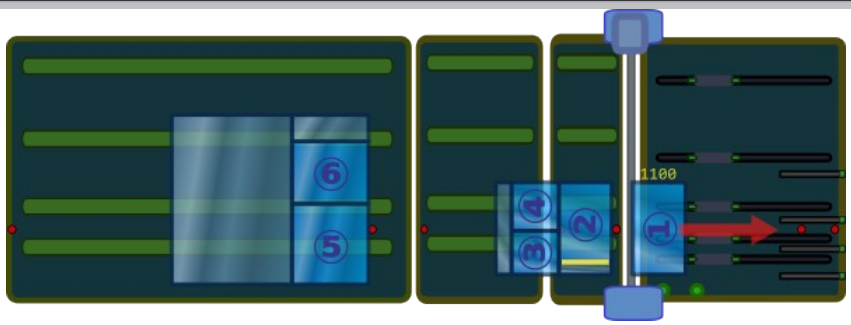


Now we have a *Subsheet* stacked in *Feed* (`stack:1`) and the previously cut *Stripe* in the processing zone (`proc:1`), that must to be aligned to `1600` where just a *score* is made from ordinate `1100` to `2200`: this isn't a complete cut but a *pre-score* of a nested cut. The *Job* creator decided that in this case pre-scoring is better than rotating and cutting normally (probably the operator has enabled this functionality somewhere in the *HMI* configuration).

The pre-score is not done for the Z cut between *Pieces* `3` and `4` probably because the width of *Piece* `3` exceeds the threshold configured to activate the pre-scores (see sect. 7.9.9 below on p.101).

***Step* 3**

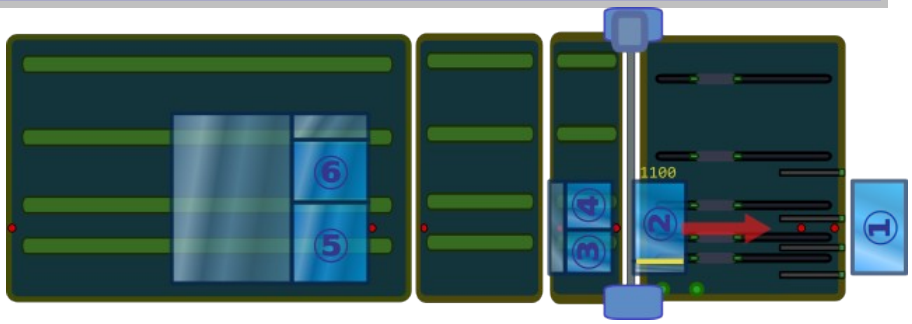
```
3="Y 1100", status:{stack:1 proc:1}, sheet:{size:1800x3210 rotate:-90 align:1100}, op:{full-cut},
  prod:{size:1100x1800 piece:1}, remn:{size:2110x1800}
```



The previously pre-scored *Stripe* is the *Subsheet* to be processed. It's already in the processing zone (`proc:1`) but needs to be rotated (`rotate:-90`) before proceeding with the cut at `1100`. This cut produces the *Piece* `1`, which is delivered to *Line-end* and retrieved, while the *Remnant* stays in the processing zone for the next operations.

***Step* 4**

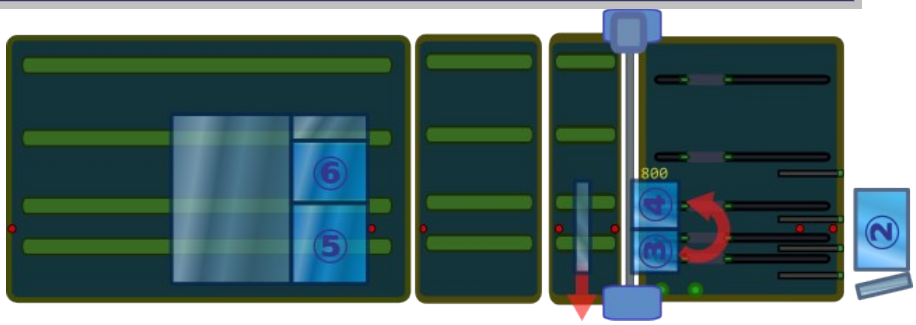
```
4="Y 1100", status:{stack:1 proc:1}, sheet:{size:2110x1800 align:1100}, op:{full-cut},
  prod:{size:1100x1800 piece:0|2}, remn:{size:1010x1800}
```



The *Remnant* from the previous *Step* is aligned at `1100` and cut. The *Product* is the previously pre-scored agglomerate containing the *Piece* `2` (`piece:0|2`), which is delivered to the operator that will proceed to manually finish the cut. The *Remnant* stays in the processing zone for the next operations.

***Step* 5**

```
5="Y 800", status:{stack:1 proc:1}, sheet:{size:1010x1800 align:800}, op:{full-cut},
  prod:{size:800x1800 rotate:-90}, remn:{size:210x1800 scrap}
```



The *Remnant* from the previous *Step* is aligned at `800` and cut. The *Product* must be immediately rotated (`rotate:-90`) and will stay there for further processing, while the *Remnant* is a *Scrap* (`scrap`) that is dumped at *Detach* side.

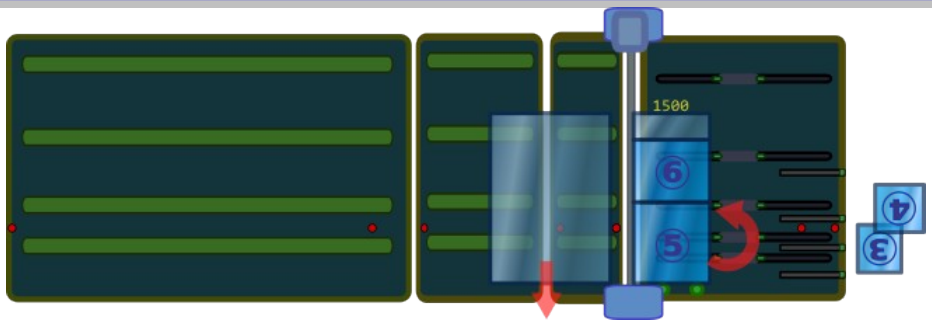
***Step* 6**

```
6="Z 800", status:{stack:1 proc:1}, sheet:{size:1800x800 align:800}, op:{full-cut},
  prod:{size:800x800 piece:3}, remn:{size:1000x800 piece:4}
```

The *Product* rotated in the previous *Step* is aligned at 800 and cut, resulting in two finished *Pieces*, *Piece* `3` on *Align* and *Piece* `4` on *Detach*. Both are delivered to *Line-end*, freeing up the processing zone.

***Step* 7**

```
7="X 1500", status:{stack:1 proc:0}, sheet:{size:4200x3210 extract align:1500}, op:{full-cut},
  prod:{size:1500x3210 rotate:-90}, remn:{size:2700x3210 scrap}
```



Since there's nothing in the processing zone, to prosecute the work we need to get the *Subsheet* stacked on *Feed* (`extract`). It is aligned at `1500` and cut; The *Product* is the last *Stripe* of the *Scheme*, that must be immediately rotated (`rotate:-90`). The *Remnant* is a *Scrap*, in this case it's special because constitutes the main *Scrap* of *Sheet*, it has often a big size and sometimes it's returned to the loading system to be stored and reused.

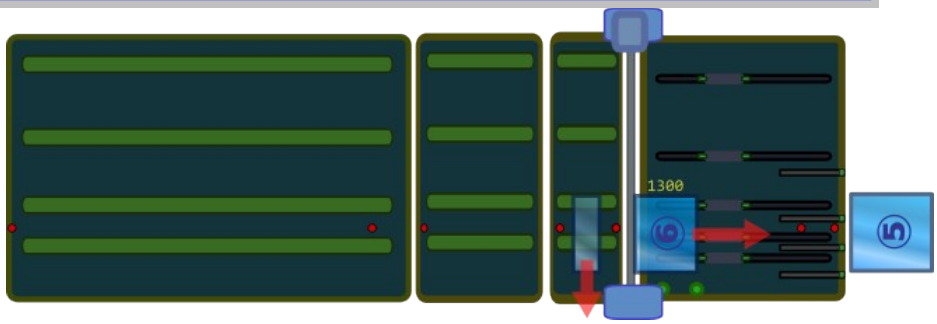
***Step* 8**

```
8="Y 1500", status:{stack:0 proc:1}, sheet:{size:3210x1500 align:1500}, op:{full-cut},
  prod:{size:1500x1500 piece:5}, remn:{size:1710x1500}
```

The previously rotated *Stripe* is cut at `1500`, resulting in *Piece* `5`, which is delivered to *Line-end*.
The *Remnant* contains other cuts so stays there for further processing.

***Step* 9**

```
9="Y 1300", status:{stack:0 proc:1}, sheet:{size:1710x1500 align:1300}, op:{full-cut},
  prod:{size:1300x1500 piece:6}, remn:{size:410x1500 scrap}
```



The *Remnant* of the previous *Step* is cut at `1300`, resulting in *Piece* `6` and a *Scrap* to be dumped.
The *Scheme* is now fully processed.

Low-E material

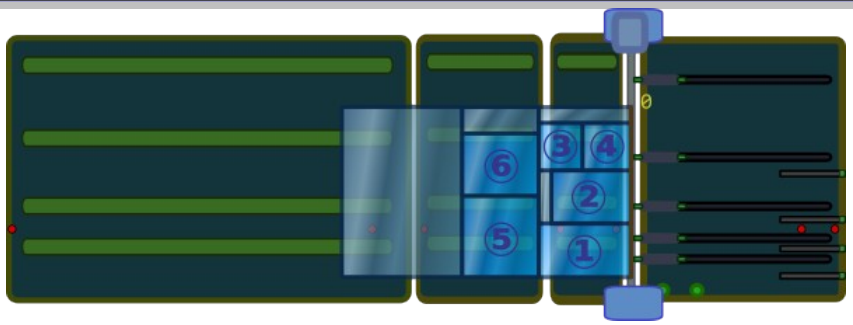
Let's explore what happens to the *Steps* for the preceding *Scheme* changing the material to low-E, which requires deleting the coating from all sides of the *Pieces*.

The number of *Steps* increases slightly due to the low-E deletion on the original *Sheet* edges (also called “head deletion”).

```
[steps]
1="head-lowe", status:{stack:1 proc:0}, sheet:{size:6000x3210 extract align:0}, op:{lowe}
2="X 1800", status:{stack:0 proc:1}, sheet:{size:6000x3210 align:1800}, op:{full-cut}, prod:{size:1800x3210}, remn:{size:4200x3210 insert}
3="Z 1600 (pre)", status:{stack:1 proc:1}, sheet:{size:1800x3210 align:1600}, op:{lowe,score y:1100|2200}
4="head-lowe", status:{stack:1 proc:1}, sheet:{size:1800x3210, rotate:-90 align:0}, op:{lowe}
5="Y 1100", status:{stack:1 proc:1}, sheet:{size:3210x1800 align:1100}, op:{full-cut}, prod:{size:1100x1800 piece:1}, remn:{size:2110x1800}
6="Y 1100", status:{stack:1 proc:1}, sheet:{size:2110x1800 align:1100}, op:{full-cut}, prod:{size:1100x1800 piece:0|2}, remn:{size:1010x1800}
7="Y 800", status:{stack:1 proc:1}, sheet:{size:1010x1800 align:800}, op:{full-cut}, prod:{size:800x1800, rotate:-90}, remn:{size:210x1800 scrap}
8="Z 800", status:{stack:1 proc:1}, sheet:{size:1800x800 align:800}, op:{full-cut}, prod:{size:800x800 piece:3}, remn:{size:1000x800 piece:4}
9="X 1500", status:{stack:1 proc:0}, sheet:{size:4200x3210 extract align:1500}, op:{full-cut}, prod:{size:1500x3210 rotate:-90}, remn:{size:2700x3210 scrap}
10="head-lowe", status:{stack:0 proc:1}, sheet:{size:3210x1500 align:0}, op:{lowe}
11="Y 1500", status:{stack:0 proc:1}, sheet:{size:3210x1500 align:1500}, op:{full-cut}, prod:{size:1500x1500 piece:5}, remn:{size:1710x1500}
12="Y 1300", status:{stack:0 proc:1}, sheet:{size:1710x1500 align:1300}, op:{full-cut}, prod:{size:1300x1500 piece:6}, remn:{size:410x1500 scrap}
```

***Step* 1**

```
1="head-lowe", status:{stack:1 proc:0}, sheet:{size:6000x3210 extract align:0}, op:{lowe}
```

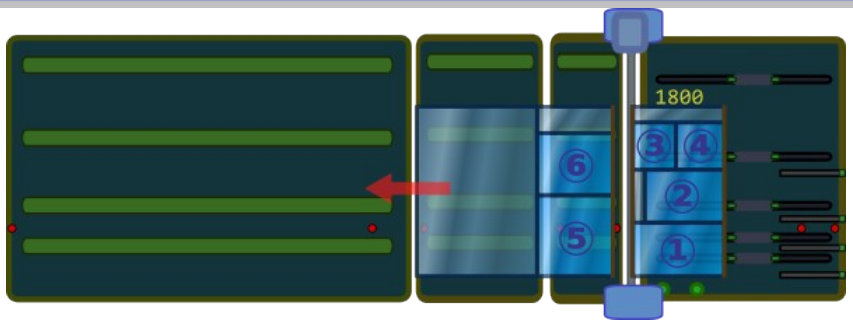


The *Sheet* needs to be extracted from *Feed* (`extract`), but before cutting the first *Stripe*, it’s necessary to delete the low-E coating from the front edge (the *Sheet* “head”).

So the *Sheet* is aligned to `0`, and is performed just the low-E deletion (`op:{lowe}`).

***Step* 2**

```
2="X 1800", status:{stack:0 proc:1}, sheet:{size:6000x3210 align:1800}, op:{full-cut},
  prod:{size:1800x3210}, remn:{size:4200x3210 insert}
```

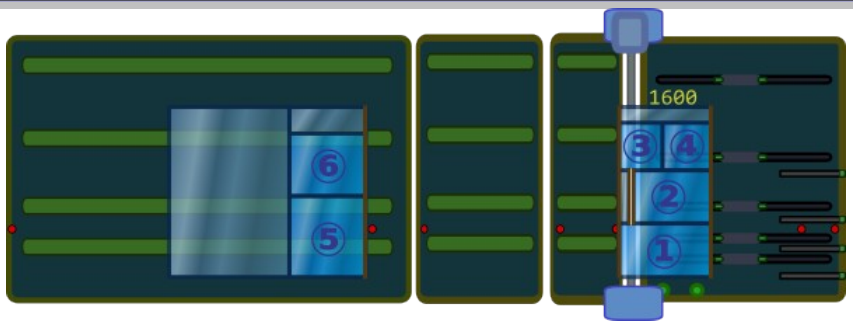


The *Sheet*, now already in processing zone (`proc:1`), is aligned to `1800` and cut. Since the material is low-E, in this context `full-cut` is equivalent to `lowe,score,open,detach`: the low-E deletion is done just before the scoring.

The *Product* remains there for further processing and the *Remnant* is reinserted back into *Feed*.

***Step* 3**

```
3="Z 1600 (pre)", status:{stack:1 proc:1}, sheet:{size:1800x3210 align:1600}, op:{lowe,score y:1100|2200}
```

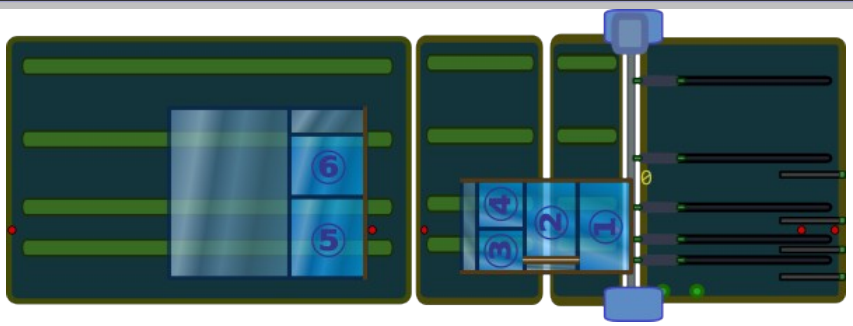


We have a *Subsheet* stacked in *Feed* (`stack:1`) and the previously cut *Stripe* ready in processing zone (`proc:1`).

As before, it has been decided to pre-score the Z cut; the only difference here is specify also a low-E deletion pass (`op:{lowe,score,y:1100|2200}`).

***Step* 4**

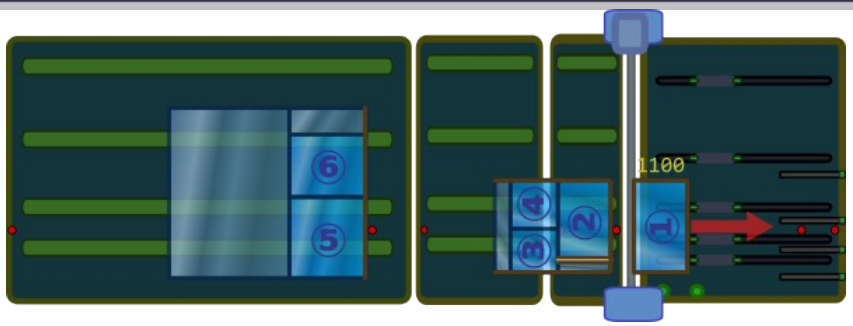
```
4="head-lowe", status:{stack:1 proc:1}, sheet:{size:1800x3210 rotate:-90 align:0}, op:{lowe}
```



The previously pre-scored *Stripe* is the *Subsheet* to process. It is already in processing zone (`proc:1`) but needs to be rotated (`rotate:-90`) before the low-E deletion on the head of *Piece* `1`.

***Step* 5**

```
5="Y 1100", status:{stack:1 proc:1}, sheet:{size:3210x1800 align:1100}, op:{full-cut},
  prod:{size:1100x1800 piece:1}, remn:{size:2110x1800}
```

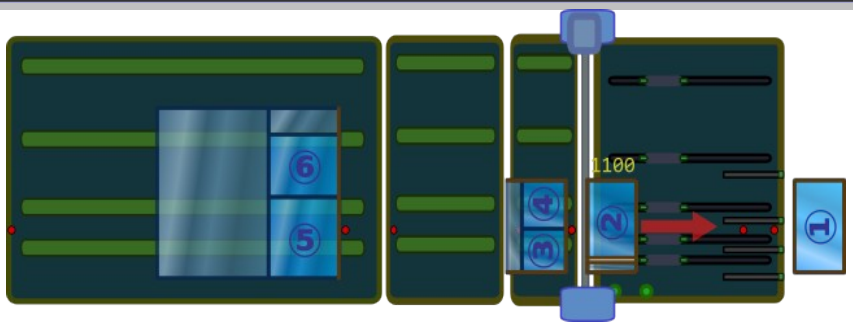


Next comes the cut at `1100`, obtaining *Piece* `1`: as you can see, all its sides have been deleted, so it can be delivered to the operator.

The *Remnant* stays there for further processing.

***Step* 6**

```
6="Y 1100", status:{stack:1 proc:1}, sheet:{size:2110x1800 align:1100}, op:{full-cut},
  prod:{size:1100x1800 piece:0|2}, remn:{size:1010x1800}
```

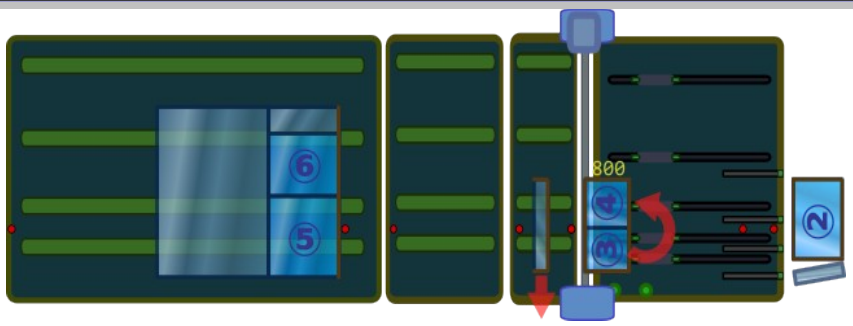


The *Remnant* of the previous *Step* is aligned at `1100` and cut, obtaining the agglomerate containing the *Piece* `2`, with all the edges deleted, which leaves the processing zone and is delivered to the operator.

The *Remnant* stays there for further processing.

***Step* 7**

```
7="Y 800", status:{stack:1 proc:1}, sheet:{size:1010x1800 align:800}, op:{full-cut},
  prod:{size:800x1800 rotate:-90}, remn:{size:210x1800 scrap}
```



The *Remnant* of the previous *Step* is aligned at `800` and cut. The *Product* is immediately rotated (`rotate:-90`), while the *Remnant* is a *Scrap* dumped at *Detach* side.

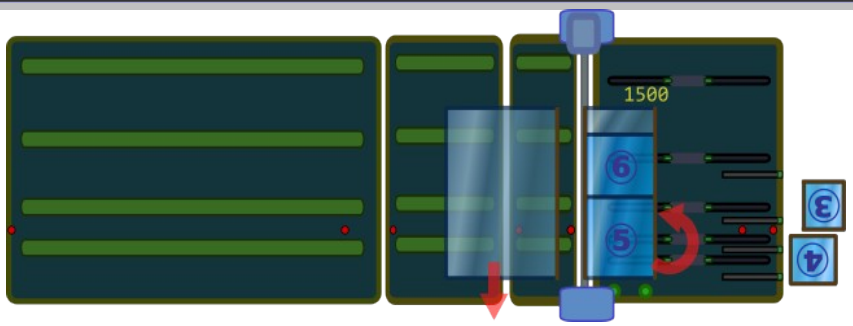
***Step* 8**

```
8="Z 800", status:{stack:1 proc:1}, sheet:{size:1800x800 align:800}, op:{full-cut},
  prod:{size:800x800 piece:3}, remn:{size:1000x800 piece:4}
```

The *Product* of the previous *Step*, just rotated, is aligned at *800* and cut; the *Product* is *Piece* `3` and the *Remnant* is *Piece* `4`, both are delivered to *Line-end*, clearing the processing zone.

***Step* 9**

```
9="X 1500", status:{stack:1 proc:0}, sheet:{size:4200x3210 extract align:1500}, op:{full-cut},
  prod:{size:1500x3210 rotate:-90}, remn:{size:2700x3210 scrap}
```



As before, to prosecute the work we need to get the *Subsheet* stacked on *Feed* (`extract`), that is then aligned at `1500` and cut; The *Product* is rotated (`rotate:-90`) and the *Remnant*, which constitutes the main *Scrap* of the *Sheet*, is removed at *Detach* side.

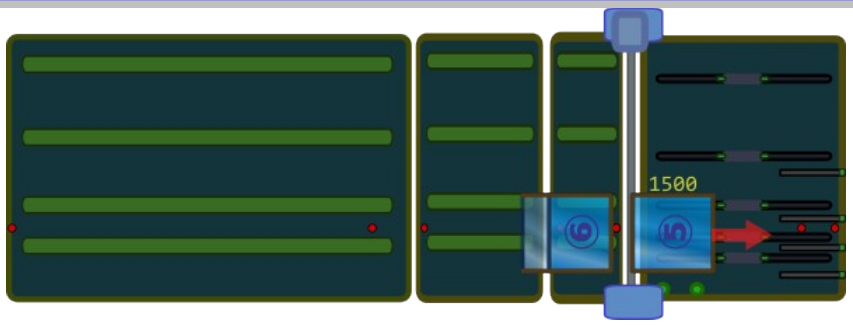
***Step* 10**

```
10="head-lowe", status:{stack:0 proc:1}, sheet:{size:3210x1500 align:0}, op:{lowe}
```

Next comes a head deletion to remove the low-E from the base of *Piece* `5`.

***Step* 11**

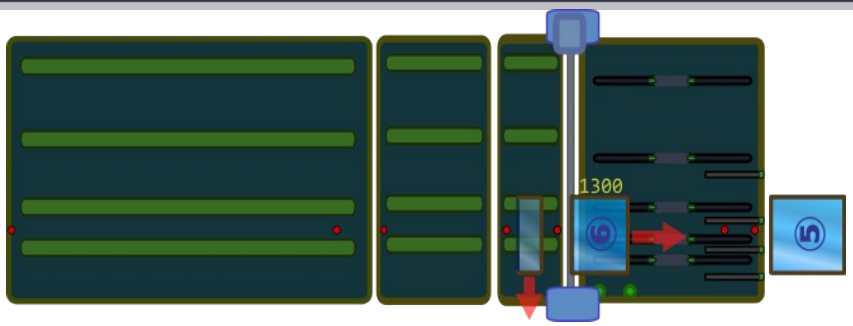
```
11="Y 1500", status:{stack:0 proc:1}, sheet:{size:3210x1500 align:1500}, op:{full-cut},
  prod:{size:1500x1500 piece:5}, remn:{size:1710x1500}
```



From now on nothing changes respect the normal processing: the cut at `1500` is done, obtaining Piece `5`, now ready, that is delivered to *Line-end*.

***Step* 12**

```
12="Y 1300", status:{stack:0 proc:1}, sheet:{size:1710x1500 align:1300}, op:{full-cut},
  prod:{size:1300x1500 piece:6}, remn:{size:410x1500 scrap}
```



The *Remnant* of the previous *Step* is cut at `1300`, obtaining *Piece* `6`.
This concludes the process for this *Scheme*.

7.9.6 Describing low-E material processing

Another example of *Steps* sequence to process a low-E *Strato* glass *Scheme*.

```
[project]
name="demo-lowe"
scheme=1/1

[scheme]
type="strato"
glass-id="44.2 BE"
size=4422x3210
sheets=1

[pieces]
; ...

[labels]
; ...

[steps]
1="X 0", status:{stack:1 proc:0}, sheet:{size:4422x3210 pop align:0}, op:{lowe}
2="X 1948", status:{stack:0 proc:1}, sheet:{size:4422x3210 align:1948}, op:{full-cut}, prod:{size:1948x3210 rotate:-90}, remn:{size:2474x3210 push}
3="Y 0", status:{stack:1 proc:1}, sheet:{size:3210x1948 align:0}, op:{lowe}
4="Y 904", status:{stack:1 proc:1}, sheet:{size:3210x1948 align:904}, op:{full-cut}, prod:{size:904x1948 piece:1}, remn:{size:2306x1948}
5="Y 1974", status:{stack:1 proc:1}, sheet:{size:2306x1948 align:1974}, op:{full-cut}, prod:{size:1974x1948 rotate:-90}, remn:{size:332x1948 scrap}
6="Z 904", status:{stack:1 proc:1}, sheet:{size:1948x1974 align:904}, op:{full-cut}, prod:{size:904x1974 piece:3}, remn:{size:1044x1974}
7="Z 904", status:{stack:1 proc:1}, sheet:{size:1044x1974 align:904}, op:{full-cut}, prod:{size:904x1974 piece:3}, remn:{size:140x1974 scrap}
8="X 2474", status:{stack:1 proc:0}, sheet:{size:2474x3210 pop align:2474}, op:{lowe}, prod:{size:2474x3210 rotate:-90}
9="Y 0", status:{stack:0 proc:1}, sheet:{size:3210x2474 align:0}, op:{lowe}
10="Y 2712", status:{stack:0 proc:1}, sheet:{size:3210x2474 align:2712}, op:{full-cut}, prod:{size:2712x2474 rotate:-90}, remn:{size:498x2474 scrap}
11="Z 1948", status:{stack:0 proc:1}, sheet:{size:2474x2712 align:1948}, op:{full-cut}, prod:{size:1948x2712 rotate:-90}, remn:{size:526x2712 scrap}
12="W 904", status:{stack:0 proc:1}, sheet:{size:2712x1948 align:904}, op:{full-cut}, prod:{size:904x1948 piece:2}, remn:{size:1808x1948}
13="W 904", status:{stack:0 proc:1}, sheet:{size:1808x1948 align:904}, op:{full-cut}, prod:{size:904x1948 piece:1}, remn:{size:904x1948 piece:1}
```

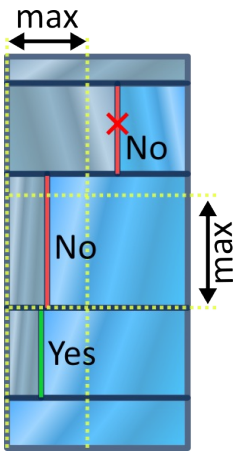
7.9.7 Describing labeling data

See the example in sect. 9.3 “Labeling Pieces of a Float Scheme on Strato machine” below on p.138.

7.9.8 Describing *Stripes* breakout

See the examples in 9.5 “Low-E deletion of a Strato Scheme on Float machine” below on p.141.

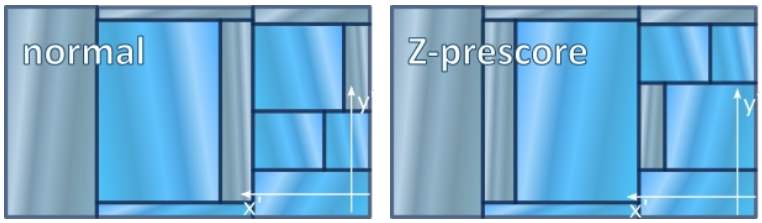
7.9.9 Describing pre-scored cuts[29]



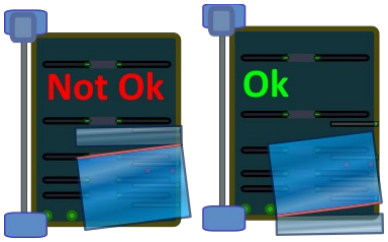
For an overview see sect. 7.5.7 “Nested cuts pre-scoring” above on p.73.

Not all nested cuts have to be pre-scored: typically, there are dimensional thresholds chosen by the operator, beyond which the cut is handled normally (the larger the glass, the more burdensome it is to separate it manually).

When pre-scoring nested cuts, the *Scheme* should be optimized to properly order the pre-scored abscissas and to place their *Scraps* towards the *Feed:*



The reason is to optimize the carriage movements and properly deliver the resulting aggregates:

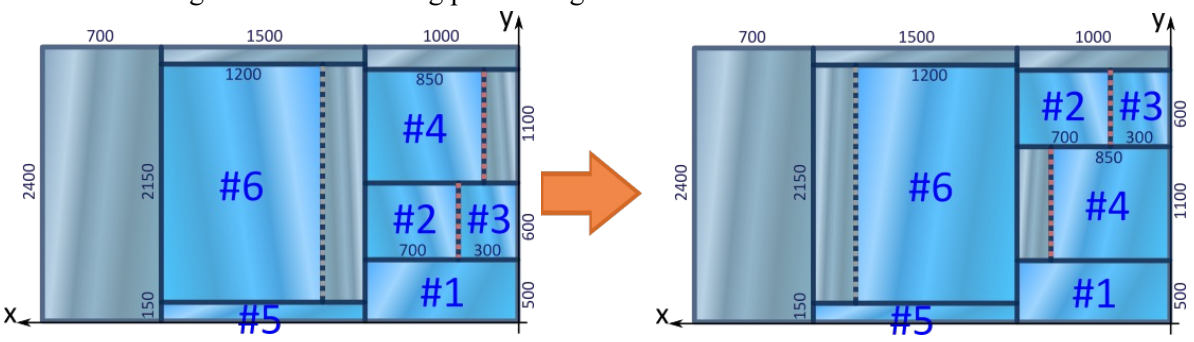


The distinctive feature of this processing is having scoring-only *Steps* between two y-coordinates (`op:{score, y:500|1600}`); after rotation, composite pieces (`prod:{piece:0|4}`) with a score still to be opened are delivered to *Line-end*, avoiding the additional rotation that would have been necessary to perform the cut automatically.

29 For further details refer to the full specification: SP062 - Preincisione dei tagli annidati.odt

Let's consider the following example: We have two *Stripes*, each containing at least one nested Z cut (dotted lines).

First, the original *Scheme* is adjusted moving the Z *Trims* toward the *Detach* so they exit downward and ordering the Z cuts in the Y direction from the first (the one with the larger X) to the last to minimize carriage movements during pre-scoring.




```
[project]
name="ZprescoreDemo"
scheme=1/1

[options]
units = { length: 0.001 }
granularity = { length: 0.1 }

[scheme]
; Optimized scheme for pre-scoring Z cuts
; The Z cuts are ordered in Y according to the pre-scoring sequence
; and Z scraps are shifted toward Detach side so they end up at the bottom
type="strato"
size=3200x2400
glass-id="33.1"
X=1000          ;     1000        1500
  Y=500 id:1    ;  +---------+-------------+-------+
  Y=1100        ;  +---+-----+----------+--+       |
    Z=850 id:4  ;  |#3 | #2  |          |  |       |
  Y=600         ;  |   |     |          |//|       |
    Z=700 id:2  ;  +---+---+-+    #6    |  |   //  |
    Z=300 id:3  ;  |       | |          |  |       |
X=1500          ;  |  #4   | |          |  |       |
  Y=150 id:5    ;  +-------+-+          |  |       |
  Y=2150        ;  |   #1    +----------+--+       |
    Z=1200 id:6 ;  +---------+------#5-----+-------+

[pieces]
1={ size:1000x500 }
2={ size:700x600 descr:"obtained with a Z cut" }
3={ size:300x600 descr:"obtained with a Z cut" }
4={ size:850x1100 }
5={ size:1500x150 }
6={ size:1200x2150 descr:"obtained with a Z cut" }

[steps]
1="X 1000", status:{stack:1 proc:0}, sheet:{size:3200x2400 pop align:1000}, op:{full-cut},
  prod:{size:1000x2400}, remn:{size:2200x2400 push}
2="Z 850 (pre)", status:{stack:1 proc:1}, sheet:{size:1000x2400 align:850}, op:{score, y:500|1600}
3="Z 300 (pre)", status:{stack:1 proc:1}, sheet:{size:1000x2400 align:300}, op:{score, y:1600|2200}
4="Y 500", status:{stack:1 proc:1}, sheet:{size:1000x2400 rotate:-90° align:500}, op:{full-cut},
  prod:{size:500x1000 piece:1}, remn:{size:1900x1000}
5="Y 1100", status:{stack:1 proc:1}, sheet:{size:1900x1000 align:1100}, op:{full-cut},
  prod:{size:1100x1000 piece:0|4}, remn:{size:800x1000}
6="Y 600", status:{stack:1 proc:1}, sheet:{size:800x1000 align:600}, op:{full-cut},
  prod:{size:600x1000 piece:2|3}, remn:{size:200x1000 scrap}
7="X 1500", status:{stack:1 proc:0}, sheet:{size:2200x2400 pop align:1500}, op:{full-cut},
  prod:{size:1500x2400}, remn:{size:700x2400 scrap}
8="Z 1200 (pre)", status:{stack:0 proc:1}, sheet:{size:1500x2400 align:1200}, op:{score, y:150|2300}
9="Y 150", status:{stack:0 proc:1}, sheet:{size:1500x2400 rotate:-90° align:150}, op:{full-cut},
  prod:{size:150x1500 piece:5}, remn:{size:2250x1500}
10="Y 2150", status:{stack:0 proc:1}, sheet:{size:2250x1500 align:2150}, op:{full-cut},
   prod:{size:2150x1500 piece:0|6}, remn:{size:100x1500 scrap}
```

Let's review the changes of this new workflow.

The normal processing would be composed by `10` *Steps* (corresponding to the `10` cuts) and involving `5` rotations for the `2` Y cuts and `3` Z cuts.

Pre-scoring processing also comprises `10` *Steps* but involves only `2` rotations, avoiding those needed for the `3` Z cuts.

In the descriptor appears the character `'|'` (`U+007C`, *vertical line*):

- In the `op{}` block are specified the Y coordinate range of the processing (ex. `Y:500|1100`) that can now be different from the glass edges
- The field `piece` in `prod{}`/`remn{}` can contain one or more integers to describe the composition of the pre-scored aggregate:
  - Ex. a *Piece* and *Scrap*: `piece:4|0` (piece `4` attached to a *Scrap* at the end)
  - Ex. Two *Pieces*: `piece:2|3` (piece `2` attached to piece `3`)
  - Ex. Arbitrary complex compositions: `piece:2|3|0|4|5|0`

A final thought: the pre-scoring of `Z 1200` in the example involves a rather large piece (`2250x1500`): not all operators would be happy to separate it manually; additionally, it is the last piece of the *Scheme*: no point in making the operator work with the machine idle.

7.9.10 Describing *Shape* processing on Strato machines

Strato machines can process *Shapes*[30], simply put their data in the `mac` descriptor in this way:

- Insert the `[shape]` sections containing the descriptors specified in sect. 5 “Polygonal shapes descriptor” on p.42 and sect. 6 “Generic shape descriptor” on p.52 (*Shape*s can be described as either polygonal or generic)

```
[shape]
name="circle1"
…
```

- In `[pieces]` section associate *Shapes* to their corresponding primitive *Pieces*

```
3={ size:1000x1200, shape:"circle1" }
```

- In `[steps]` section in `prod`/`remn` blocks indicate the *Shape* associated to the obtained *Piece*

```
prod{ piece:3, shape:"circle1" }
```

⚠ `[shape]` sections must always precede the `[steps]` section.

30 For details refer to the full specification: SP064 - Sagome generiche su laminato.pdf

```
; Strato job descriptor with shapes

[project]
name="demo-shapes"
scheme=1/1

[options]
units = { length: 0.001 }
granularity = { length: 0.001 }

[scheme]
type="strato"
glass-id="33.1"
size=6000x3210
sheets=1
X=1000          ;   Y ^              6000
  Y=1200 id:1   ;     +-----+---------+-------+-------+
  Y=900         ;     |     |---------|       |       |
    Z=500 id:2  ;     |--+--|     1500|  //   |   //  | 3
    Z=500 id:3  ;  900|#2|#3| #5      |       |       | 2
X=2000          ;     |--+--|---------|-------|       | 1
  Y=1000 id:4   ; 1200|     |     1000| #6    |       | 0
  Y=1500 id:5   ;     | #1  | #4      |   1100|       |
X=1100          ;     +-----+---------+-------+-------+  ---> X
  Y=1100 id:6   ;      1000    2000     1100

[pieces]
1={ size:1000x1200 descr:"first" shape:"diag_1" }
2={ size:500x900   descr:"second" }
3={ size:500x900   descr:"third" }
4={ size:2000x1000 descr:"fourth" shape:"arch_R700" }
5={ size:2000x1500 descr:"fifth" }
6={ size:1100x1100 descr:"sixth" shape:"circ_R500" }

[labels]
label-size=100x80
alignment=C

[shape]
name="diag_1"
options={strict:1}
size=1000x1200
1=line:{w:250,W:-300} id:"A"
1=id:"B"

[shape]
name="arch_R700"
size=1000x2000
*=path:{
    start:{from:(999, 1740.917)},
    arc+:{to:(1,1740.917) C:(500,1250)}
   }

[shape]
name="circ_R500"
size=1100x1100
*=path:{
    start:{from:(1050,550)},
    arc-:{to:(50,550) C:(550,550)},
    arc-:{to:(1050,550) C:(550,550)}
   }

[steps]
1="X 1000", status:{stack:1 proc:0}, sheet:{size:6000x3210 pop align:1000}, op:{full-cut},
  prod:{size:1000x3210 rotate:-90°}, remn:{size:5000x3210 push}
2="Y 1200", status:{stack:1 proc:1}, sheet:{size:3210x1000 align:1200}, op:{full-cut},
  prod:{size:1200x1000 piece:1 shape:"diag_1"}, remn:{size:2010x1000}
3="Y 900", status:{stack:1 proc:1}, sheet:{size:2010x1000 align:900}, op:{full-cut},
  prod:{size:900x1000 rotate:-90°}, remn:{size:1110x1000 scrap}
4="Z 500", status:{stack:1 proc:1}, sheet:{size:1000x900 align:500}, op:{full-cut},
  prod:{size:500x900 piece:2}, remn:{size:500x900 piece:3}
5="X 2000", status:{stack:1 proc:0}, sheet:{size:5000x3210 pop align:2000}, op:{full-cut},
  prod:{size:2000x3210 rotate:-90°}, remn:{size:3000x3210 push}
6="Y 1000", status:{stack:1 proc:1}, sheet:{size:3210x2000 align:1000}, op:{full-cut},
  prod:{size:1000x2000 piece:4 shape:"arch_R700"}, remn:{size:2210x2000}
7="Y 1500", status:{stack:1 proc:1}, sheet:{size:2210x2000 align:1500}, op:{full-cut},
  prod:{size:1500x2000 piece:5}, remn:{size:710x2000 scrap}
8="X 1100", status:{stack:1 proc:0}, sheet:{size:3000x3210 pop align:1100}, op:{full-cut},
  prod:{size:1100x3210 rotate:-90°}, remn:{size:1900x3210 scrap push}
9="Y 1100", status:{stack:1 proc:1}, sheet:{size:3210x1100 align:1100}, op:{full-cut},
  prod:{size:1100x1100 piece:6 shape:"circ_R500"}, remn:{size:2110x1100 scrap}
```

8 *Float* machines

8.1 Objectives

Provide the concepts and criteria regarding the generation and processing of *Schemes* in our monolithic glass cutting machines.

Clarify how the generic machine interface specializes for *Float* machines: the specific resources and the content of the `mac` files.

8.2 Context

The implementation of the interface that abstracts the interaction with a generic *Float* machine was implemented subsequently the one of the *Strato* machines and happened much later than the design of the machines themselves.

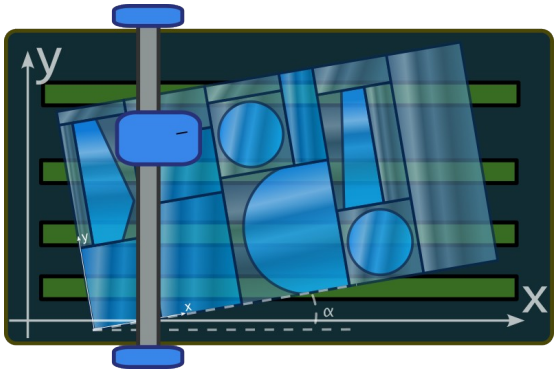
For the basics of *Float* machines, please refer to sect. 2.5 “Float machines” above on p.13.

8.3 Reference system

The machine's reference system consists of a pair of orthogonal X and Y axes, with X parallel to the base and Y to the height of the *Sheet*.

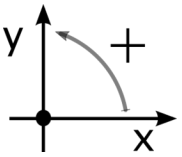
The position reference point usually refers to the primary scoring head, while other tools (such as low-E wheel, secondary head, TPF scraper, label applicator, marker, etc.) are characterized by their offset relative to it.

The reference system related to the sheet is similar: the vertex of the sheet corresponding to zero is called the reference vertex, and it matches the one closest to the machine's zero point.



The positive direction of angles is the rotation that moves the X-axis to overlap with the Y-axis.

In a right-handed system, this is counterclockwise, while in a left-handed system, it's clockwise



Some machines are built flipped, with a reversed X-axis; this affects only the graphic representation of the machine and the *Schemes* processed by it.

8.4 *Schemes* for *Float* machines

From a very general standpoint a *Float Scheme* may be represented by a set of tool paths applied on a rectangular *Sheet* of known dimensions.

As anticipated in sect. 2.3 “Cuts tree/Scheme basics” above in p.12, a glass cut must span edge to edge, so we divide those paths into two categories: rectilinear/straight cuts and shaped cuts.

The former consist of cuts parallel to the *Sheet*’s sides, passing through the entire glass (that’s why they’re sometimes called *cross-cuts*), to create openable sections and divide it into smaller rectangles, the *Pieces*/*Scraps*; the latter are generic paths within their respective *Primitive* rectangles, obtaining *Shapes*.

This division is useful for technical reasons in glass cutting:

- The processing of rectilinear/straight cuts always precedes that of shaped cuts, the main reason is related to scoring, to prevent any fracture from spreading outside the *Primitives*
- The speed and push force of the tools generally need to be different on curvilinear paths
- The orientation of the scoring head is managed differently (modulo 180° for rectilinear cuts)

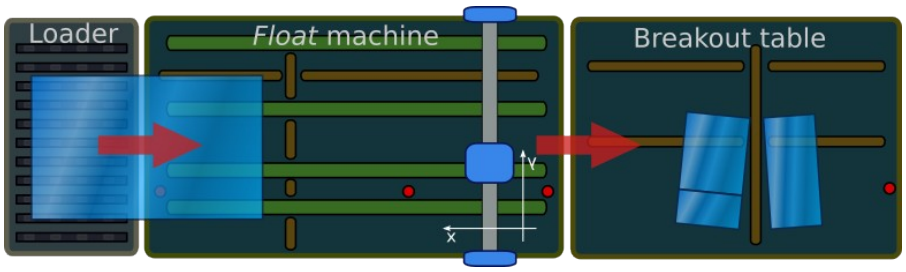
8.5 Job generation rules

Given the *Sheet*, the tools, and the *Pieces* to be obtained, the *HMI* generates the Job for the machine. The rules for generating a *Float Job* are well-established, so we will leverage the current practices without formalizing anything, for now.

8.6 Typical workflow

To clarify, here’s the typical workflow:

- A *Project* is generated and transmitted
- A *Scheme* is selected and sent (synchronized)
- The corresponding (dimensions, material) *Sheet* is loaded onto the machine main table
- The *Scheme* is processed by the machine:
  - If the machine lacks mechanical references, it will detect the actual position of the *Sheet* to perform the cuts relative to it
  - If a labeling system is available and enabled, the pieces are labeled
  - If the material has TPF or low-E, they are removed from the cut path before scoring
  - Scoring is performed
- The *Sheet* may be moved downstream to an external breakout table to open the cuts
- The operator opens the cuts using the breakout bars, assisted by the cuts pattern displayed on a TV set



8.7 Float machine characteristics

On connection (see sect. 3.6.1 “Greet message” above on p.25) the machine transmits the following specific parameters[31] that characterize its features:

`mach-capabilities`	`probe` — Equipped with glass search probe `probe2` — Equipped with secondary glass search probe `lowe` — Equipped with low-E wheel `tpf` — Equipped with *TPF* scraper `label` — Equipped with a *Piece* labeling system `brkbar-bridge` — Equipped with a breakout bridge for *Stripes* `shift` — Capable of moving *Sheets* `shapecurv` — Capable to process generic *Shapes*	
`cut-table-width`	Overall width of the machine main table	*useful for visualization*
`cut-table-height`	Overall height of the machine main table	
`x-min, x-max, y-min, y-max`	Reachable area by the primary scoring head	
`*-offset`	Tools relative position respect the primary scoring head	
`score-edges-margin`	Minimum distance of scoring from the glass edges	
`score-cross-margin`	Minimum distance of scoring from intersections	
`lowe-width`	Transverse size of the low-E wheel	`lowe ∈ mach-capabilities`
`lowe-margin`	Minimum longitudinal distance from the glass edges	
`tpf-width`	Transverse size of the TPF scraper	`tpf ∈ mach-capabilities`
`tpf-margin`	Minimum longitudinal distance from the glass edges	
`thickness-max`	Maximum supported glass thickness	
`latches-queue-max-size`	Maximum storable latches for each movement in shape scanning (interactive mode)	

31 For details refer to `Interface.xml` *file*

8.8 Material parameters

A summary table of material processing parameters[32]:

	sheet-type	*Material type*	float/strato, lowe, filmsup, filminf (float/strato mutually exclusive)
	h-glass	*Glass thickness*	[mm]
	cut-recipe	*Cut sequence recipe*	standard
	wheel-angle	*Scoring wheel gradation*	[deg]

	f-score-init-1	*Initial pushing force of scoring head 1*	[N]
	f-score-rect-1	*Rectilinear pushing force of scoring head 1*	[N]
	f-score-curv-1	*Curvilinear pushing force of scoring head 1*	[N]
	v-score-rect-1	*Rectilinear speed of scoring head 1*	[m/min]
	v-score-curv-1	*Curvilinear speed of scoring head 1*	[m/min]

	f-score-init-2	*Initial pushing force of scoring head 2*	[N]
	f-score-rect-2	*Rectilinear pushing force of scoring head 2*	[N]
	f-score-curv-2	*Curvilinear pushing force of scoring head 2*	[N]
	v-score-rect-2	*Rectilinear speed of scoring head 2*	[m/min]
	v-score-curv-2	*Curvilinear speed of scoring head 2*	[m/min]

32 For details refer to Interface.xml *file*

	f-lowe	*Pushing force of the low-E coating deletion wheel*	[N]
	v-lowe	*Speed of low-E coating deletion*	[m/min]
	f-tpf	*Pushing force of TPF scraper*	[N]
	v-tpf	*Speed of TPF scraper*	[m/min]

	h-brkbar	*Breakout bar opening height*	0:auto 1:low 2:high
	t-brkbar	*Breakout bar opening time*	[s]
	f-clamps-brk	*Clamps push during breakout bar*	[N]

The *HMI* can open a dialog to edit these parameters as described in sect. 3.15 “Editing material parameters” above on p.33.

8.9 Communicating with *Float* machines

The communication is carried out as described in sect. 3 “Communication protocol” above on p.18.
Let's look at the specific operations for *Float* machines.
As always, please refer to the file `Interface.xml` for the meaning and interpretation of the fields.

8.9.1 Monitoring the *Float* machine status

Regarding how to monitor the machine status, see sect. 3.10 “Monitoring the machine status” above on p.28.
The *Float*-specific monitored fields, added to the common ones listed in sect. 3.10.4 “Common monitored fields” above on p.30, are:

`status-lights`	Machine status lights
`work-selectors`	Work selector status
`spdovd-lowe`	Speed partialization during low-E deletion
`probe-status`	Current reading of the selected glass probe (`mach-capabilities` has `probe`)

These fields, along with `spdovd`, `msg-list` and `emg-list`, should be constantly displayed via the graphical *widgets* representing them.

For displaying the *Scheme* during breakout (*monitor TV*):

`ebrk-scheme`	Index of *Scheme* currently in breakout
`ebrk-stripe`	Index of *Stripe* currently in breakout (automatic breakout bridge)
`ebrk-scheme-done`	*Scheme* breakout completed

To visualize the status, the following can also be useful:

`x-pos, y-pos, z-pos`	Current position of the axes respect the table

During processing:

`x-tool-pos, y-tool-pos, z-tool-pos`	Current tool position relative to the *Scheme*
`step`	Index of the current operation in the *Scheme* (*implementation-defined*), limited by `steps-count`

Significant events

`scheme-done=0➞1 and can-receive-job=1`	*Scheme* completed	Synchronize the next if present
`can-receive-job=0➞1 and scheme-done=1`		
`ebrk-scheme=n➞m`	New *Scheme* in breakout	Update the displayed *Scheme* (none if zero)
`ebrk-stripe=n➞m`	New *Stripe* in breakout	Highlight the *Stripe* currently in breakout
`ebrk-scheme-done=0➞1`	Breakout of *Scheme* completed	May proceed with the next *Project*

Starting from version `2025-06` of MacoLayer, the `msg="event"` notification is sent to signal a work event. See sect.3.11 “Events notification” above at p.31

8.9.2 Graphic Widgets

The *HMI* will feature graphical controls (*widgets*) to display the current status and allow the operator to modify the machine's behavior.

Machine status lights

The `status-lights` resource, included in `$status`, collects read-only boolean values indicating some machine's status flags. These are typically represented graphically with stylized icons, similar to car dashboard lights:

`cnc-error`	Machine *Controller* in error
`air-ok`	Air pressure present/Pneumatic actuators ready
`power-on`	Machine powered
`parked`	Axes in rest position
`ready-to-work`	Machine ready with a *Job* uploaded/synchronized (waiting *start* button to proceed)
`may-enter`	It is possible to approach the machine

Understanding the semantic meaning of these signals is important for associating appropriate explanatory images, which are the responsibility of the *HMI* for stylistic reasons; for texts/translations, it is possible to refer directly to the content of `Interface.xml`.

Speed override

Due to historical reasons, the general speed override percentage (`spdovd`) has no effect on the low-E deletion phase, so a specific override (`spdovd-lowe`) is added for that.

Widgets like knobs and edits allow users to adjust an integer value from `0` to `100`. When a value change event occurs, a write request is sent. For instance:

⇨ `[SOH]id=304[STX]spdovd=95[ETX]`

Scoring heads selection

For machines equipped with dual scoring heads[33], it is possible to select which scoring heads to use by adding a string to `work-selectors` value (described on the next page). There are three possible states:

- `<empty>` (*default: use primary head only*)
- `T2-only` (*use secondary head only*)
- `T1+2` (*use primary head for rectilinear cuts and secondary head for shapes*)

You must use a three-state selector. Each click moves to the next state. For example, clicking three times in a row will sequentially send the requests:

⇨ `[SOH]id=305[STX]work-selectors=…,T2-only[ETX]`

⇨ `[SOH]id=306[STX]work-selectors=…,T1+2[ETX]`

⇨ `[SOH]id=307[STX]work-selectors=…[ETX]`

And so on, in a cyclical manner. Replace the ellipses with the current status of the other work selectors.

33 `mach-capabilities` contains `t2`

Work selectors

The `work-selectors` resource, included in `$status`, gathers the operator-adjustable selectors that affect the processing. Some of these are displayed only if the machine has certain capabilities (refer to the `mach-capabilities` column):

		`mach-capabilities`
`enab-tools`	Enable/disable tools	
`enab-lowe`	Enable/disable low-E wheel	`lowe`
`cut-lub`	Activates scoring heads lubrication	
`enab-labels`	Enable piece labeling	`labels`
`no-safety`	Disable tool outside glass safety	
`man-speed`	Force processing speed	
`flip-x`	Mirrored processing	
`park2`	Select secondary rest position of axes	
`no-search`	Do not detect sheet position	`probe`
`probe2`	Use secondary glass search probe	`probe2`
`test-size`	Enable actual sheet size check	`probe`
`test-defects`	Enable sheet defects check	`probe`
`confirm-start`	Request confirmation to start processing	
`auto-start`	Enable automatic job start after sheet load	`shift`
`auto-load`	Enable sheet automatic load on table	`shift`
`auto-moveout`	Enable automatic move-out after job	`shift`
`auto-lowerect`	Enable low-E wheel automatic rectification	`lowe`
`post-proc`	Enable the automatic breakout/labeler bridge	`brkbar-bridge`
`T2-only/T1+2`	Select scoring heads	`t2`

Understanding the semantic meaning of these selectors is important for associating appropriate explanatory images, which are the responsibility of the *HMI* for stylistic reasons; for texts/translations, it is possible to refer directly to the content of `Interface.xml`.

As one or more selectors change, a request is sent to write the overall state, for example, if `enab-tools` and `cut-lub` were already active, on enabling `auto-start`:

⇨ `[SOH]id=308[STX]work-selectors=enab-tools,cut-lub,auto-start[ETX]`

The click event NEVER changes the widget state; it only generates a write request. The state will ONLY be set during the next reading of the actual value, performed through monitoring (see sect. 8.9.1 above).

8.9.3 Machine visualization

A view of the machine can be created using the following data communicated on connection[34]:

`cut-table-width`	Overall width of the machine main table
`cut-table-height`	Overall height of the machine main table
`x-min, x-max, y-min, y-max`	Reachable area by the primary scoring head

34 See sect. 8.7 “Float machine characteristics” above on p.107

8.9.4 Generating a *Float Project*

The considerations mentioned in the sect. 3.12 “Generating a Project” above on p.32 apply.

The `mac` files that describe a *Job* for *Float* machines, described in sect. 8.10 “Float Scheme descriptor (mac)” below on p.121, usually contain a `[paths]` section, describing the operations to process the *Scheme*.

8.9.5 Project transmission

See sect. 3.13 “Project transmission” above on p.32.

Essentially once the `mac` files have been generated in a certain folder, the *Project* is transmitted/uploaded with the command similar to:

⇨ `[SOH]id=1432[STX]prj-name="demo1";prj-path="C:\folder"[ETX]`
⇦ `[SOH]rep-to=1432[ETX]`

8.9.6 Scheme synchronization

After a *Project* transmission or navigation (browsing the *Schemes*), the only remaining action to start working is the so called *synchronization*, which involves selecting the desired *Scheme* to be processed.

It consists of sending a request like:

⇨ `[SOH]id=1433[STX]mode="auto";scheme=4;sheet=3;step=1;glass-id="FL4";`
`sheet-type="float"; h-glass=4; [ETX]`
⇦ `[SOH]rep-to=1433[ETX]`

> ⚠ Always specify the material in `glass-id` to ensure the correct parameters. To enable consistency checks for the material the optional fields `sheet-type` and `h-glass` can be specified.

> ℹ The *sheet* field is optional and represents the *Scheme* repetition.

If the response is positive, the machine is ready to work the selected job (in jargon, it’s *synchronized* with the *HMI*); if you want to be sure, you can check that the `job-loaded` field is active (monitored in `$status`).

Automatic selection of mirrored processing

When a *Strato* material is selected[35], the operator must execute the cuts again on the lower glass layer after flipping the *Sheet* and selecting the mirrored cut (`flip-x`).

The *HMI* may automatize these operations adding a second repetition in case *Strato* material and enabling the `flip-x` work selector on the synchronization of the second repetition:

⇨ `[SOH]id=1515[STX]mode="auto"; scheme=1; sheet=2; glass-id="33.1";`
`work-selectors=…,flip-x[ETX]`

Mirroring can also be enabled separately with a dedicated message:

⇨ `[SOH]id=1516[STX]work-selectors=…,flip-x[ETX]`

Replace the ellipses with the current status of the work selectors.

> ℹ The *HMI* does not need to reset the `flip-x` selector: at the end of the operation, the machine or the operator will handle its reset.

35 `glass-id` is a *Strato* material or, if the *HMI* is handling the material parameters, `sheet-type` contains `strato`

8.9.7 During automatic work

Once a *Job* is *synchronized* on the machine, the operator can start the process of the selected *Scheme*. When the `scheme-done` signal is raised, the *Scheme* has been fully processed.

For graphical feedback, the *HMI* can use `scheme-progress` and `scheme-remaining-time` to display the completion status.

The *HMI* may show the current tool (`op-status`) and its position relative to the *Scheme/Sheet*:

`x-tool-pos, y-tool-pos, z-tool-pos`	Current position of the tool relative to the *Sheet*

…and use this data to perhaps color the processed paths of the *Scheme*.

The following `op-status` values are guaranteed for *Float* machines:

`probing`	Searching/detecting the *Sheet* position
`scraping`	Removing the protective film (*TPF*) with the scraper
`deleting`	low-E deletion
`scoring`	Etching the glass
`working`	Other processing (marking, labeling, …)

Additionally, the *Scheme*/*Sheet* can be visualized relative to the table reference system using the following data:

`cut-table-width, cut-table-height`	Table dimensions	Communicated on connection
`-`	Table position	Unknown, set the origin so that the reachable area is within the table
`x-min, x-max, y-min, y-max`	Reachable area	Communicated on connection
`*-offset`	Tools relative positions	Communicated on connection, with them it's possible to calculate the position of the current tool and the actual workable area (reachable by all involved tools)
`x-pos, y-pos, z-pos`	Axes position (relative to table)	Monitored in `$status`, the reference tool is the primary scoring head
`sheet-position, sheet-angle`	*Sheet* location on table	Communicated by the machine after glass detection, it assumes a rectangular *Sheet*
`sheet-size`	*Sheet* dimensions	Already known to the *HMI*

8.9.8 Clearing a job on the machine

Send a message as described in sect. 3.17 “Job desynchronization” above on p.35.

The effect is to invalidate the current selected *Scheme*, thus inhibiting any machine actions if the start button is pressed.

8.9.9 Label printing requests

The *HMI* must be ready to respond to any requests from the machine as described in sect. 3.14 “Serving prints” above on p.33.

8.9.10 Measurement notifications

Some machines, if equipped with the glass search probe (`mach-capabilities` contains `probe`), might send notifications containing measured data.

Sheet size detection

The operator can request the machine to measure the actual dimensions of the *Sheet* on the table; when this happens, the *HMI* may receive a notification like this from the machine[36]:

⇦ `[§]msg="data"[§]sheet-size=2522.34x798.851[ETX]`

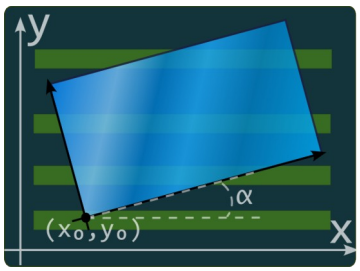
> If a dialog for optimizing a new *Sheet* is shown, it would be convenient to transcribe these actual dimensions into it.
>
> Alternatively, a '*measure sheet from machine*' function could be implemented, where the system waits for this notification and then uses the data for optimization.

Sheet actual position detection

During the processing of a *Scheme*, when the machine detects the actual position of the *Sheet*, it may send something like[36]:

⇦ `[§]msg="data"[§]sheet-position=(100.5,23.34);`
`sheet-angle=5.327[ETX]`

Which describes the actual position of the *Sheet* on the table as shown in the figure, assuming it is rectangular and has the dimensions specified in the *Scheme* data.



36 As always refer to `Interface.xml` to interpret these data.

8.9.11 Shape scanning (interactive mode)

An *interactive* mode is available, allowing the *HMI* to move the axes and use the glass probe (currently the only tool available for this mode).
This mode is introduced for the *shape scanning* function.

Commands

`command`	`line`	Detect with `probe` along a straight line	
	`arc+`	Detect with `probe` along a positive arc (counterclockwise in right-handed systems)	
	`arc-`	Detect with `probe` along a negative arc (clockwise in right-handed systems)	
	`move`	Fast free movement	
`to`	Coordinates of the movement endpoint		`(x,y)`
`center`	Coordinates of arc's center		`(x,y)`
`from`	(Optional) Coordinates of the starting point of the movement		`(x,y)`
`edge-type`	Desired event for position capture/latch		`rise`, `fall`, `both`
`speed-perc`	Speed/acceleration partialization of the movement		[%]

The `line`, `arc+`, `arc-` commands are tool-controlled movements; in this case the only one supported is the `probe` that detects material on the table, which allows sampling the edge points of a template.
For these movements it is possible to specify the type of edge sought (`edge-type`) and the speed of detection (`speed-perc`).

> If `edge-type` or `speed-perc` are not specified, the last value is used.
> The initial values are: `edge-type=rise` and `speed-perc=100`

`edge-type` determines the event that triggers the capture (*latch*) of the probe's position, collecting points where the probe reading changes. You can select rising, falling, or both edges.

The coordinates always refer to the tool's position, specifically the `probe` moving along the search paths. Each movement requires at least an endpoint (`to`) be specified; arcs also require the center coordinates (`center`).
The starting position defaults to the current tool position (`x-tool-pos`, `y-tool-pos`) unless starting point coordinates (`from`) are explicitly specified. This flexibility means the move command is seldom required, though it remains available for general purposes.

Adding the suffix `.search` to the command indicates an intention to stop the movement immediately at the first capture.

> In case of short movements, this might slow down operations rather than speeding them up: use it when scanning for the first point, not while sampling the next adjacent point.

Once the command is transmitted, the machine will acknowledge it with an immediate reply: if positive, the machine will proceed with execution and notify the result upon command completion.

> The command should be considered executed only upon receiving the notification of its completion result; no further commands can be sent until the previous one is fully completed.

Commands result

`latches-count`	Number of captures collected during the movement	`>=0`
`latch-<n>`	Data of the collected n-th latch (`1÷latches-count`)	`r|f(x,y)`
`probe-status`	Current reading of the selected glass probe	`0`:*not reading*, `1`:*reading*
`x-tool-pos`, `y-tool-pos`	Probe current position	`x-pos`,`y-pos` shifted by `probe-offset`

Upon completing the command execution, a notification will be received containing the fields listed in the table above.
The `latches-count` field indicates the number of latches occurred. A *latch* is the probe position recorded at a specified reading change (rise or fall).
For each capture, the notification will contain a field like `latch-<num>=<edgetype>(x,y)`, where the captured coordinates are preceded by a character indicating the type of edge (`r`=*rise*, `f`=*fall*) for example:

`latch-1=r(10,-20)`: first latch on a rise edge at `x=10 y=-20`
`latch-2=f(40,-80)`: second latch on a fall edge at `x=40 y=-80`

These fields are chronologically ordered: `latch-1` is the first latch and `latch-n` is the last if `latches-count=n`.

The `latches-queue-max-size` field, communicated on first connection, declares how many points the machine can collect during a movement: if the number of latches is equal to this value (`latches-count==latches-queue-max-size`), it's not excluded that there may have been further latches along the rest of the path.

If the `latches-count` field has a positive value `n`, it means that there have been latches: in this case, is guaranteed the presence of the fields `latch-1`, … , `latch-n`.

Other useful data

To determine the search area and movement targets, consider the following data[37], communicated on connection:

`x-min, x-max, y-min, y-max`	Reachable area
`probe-offset`	Relative position of the probe
`latches-queue-max-size`	Maximum number of latches that can be stored per movement
`mach-capabilities`	It should contain *probe*, otherwise the machine cannot perform shape scanning

The reachable area of the probe can be determined by algebraically adding its offset to the axes limits:

`x-min + offset.x` `x-max + offset.x`
`y-min + offset.y` `y-max + offset.y`

The machine automatically constrains commanded movements to stay within these limits.

Although uncommon, it is possible to read on demand:

`probe-status`	Current reading of the selected glass probe

37 See sect. 8.7 "Float machine characteristics" above on p.107

Sequence

It's possible to request the machine to enter interactive mode sending:

⇨ [%]id=10[§]mode="interactive";tool="probe"[£]
⇦ [%]rep-to=10[§]

If the reply is positive, wait for the `mode` (monitored in `$status`) to become `"interactive"`, otherwise probably the machine is busy and not available, in this case wait for the conditions to retry (`mach-status="ready"`).

In interactive mode, movement commands can be sent. Once the machine receives the command positively, wait for a notification that confirms its completion and result.

> The machine is ready for interactive mode when `mach-status="ready"`. The intent to perform shape scanning is expressed with `tool="probe"`. Naturally, the machine must be equipped with a search probe (`mach-capabilities` must include `probe`).

Example searching along a **straight line**:

⇨ [%]id=11[§]command="line";from=(0,0);to=(2000,1000);edge-type=both;speed-perc=100[£]
⇦ [%]rep-to=11[§]

Both edges are searched, and the probe will reach the endpoint of the movement.

On completion the machine may send a notification like this:

⇦ [%]msg="done 11"[§]latches-count=2;latch-1=r(200.4,100.2);latch-2=f(1603.2,801.6);
probe-status=0;x-tool-pos=2000.001;y-tool-pos=1000[£]

The notification indicates there were two[38] latches: the first on a rising edge followed by another on a falling edge; at final position the probe is not reading anything.

If there had been no latches, the notification would have been:

⇦ [%]msg="done 11"[§]latches-count=0;probe-status=0;x-tool-pos=2000.001;y-tool-pos=1000[£]

Example searching along a **straight line with *stop***:

⇨ [%]id=12[§]command="line.search";from=(0,0);to=(2000,1000);edge-type=rise[£]
⇦ [%]rep-to=12[§]

This time, the request is to stop at the first latch on a rising edge; in the event of a latch a potential notification might be

⇦ [%]msg="done 12"[§]latches-count=1;latch-1=r(200.4,100.2);probe-status=1;
x-tool-pos=350.462;y-tool-pos=175.231[£]

So a single capture on the rising edge.

In the absence of captures, the notification would be similar to the previous example

Example searching along an **arc**:

⇨ [%]id=13[§]command="arc+";from=(120,200);to=(180,200);
center=(150,200);edge-type=rise,fall;speed-perc=50[£]
⇦ [%]rep-to=13[§]

> The following notations are equivalent:
> `edge-type=rise,fall`
> `edge-type=both`

Aside from specifying the center, everything the rest remains the same, including command completion notifications.

Example of **free movement**:

⇨ [%]id=20[§]command="move";to=(0,200)[£]
⇦ [%]rep-to=20[§]
...
⇦ [%]msg="done 20"[§]latches-count=0;probe-status=0;x-tool-pos=0.001;y-tool-pos=199.998[£]

On command completion, the status and actual position of the probe are notified.

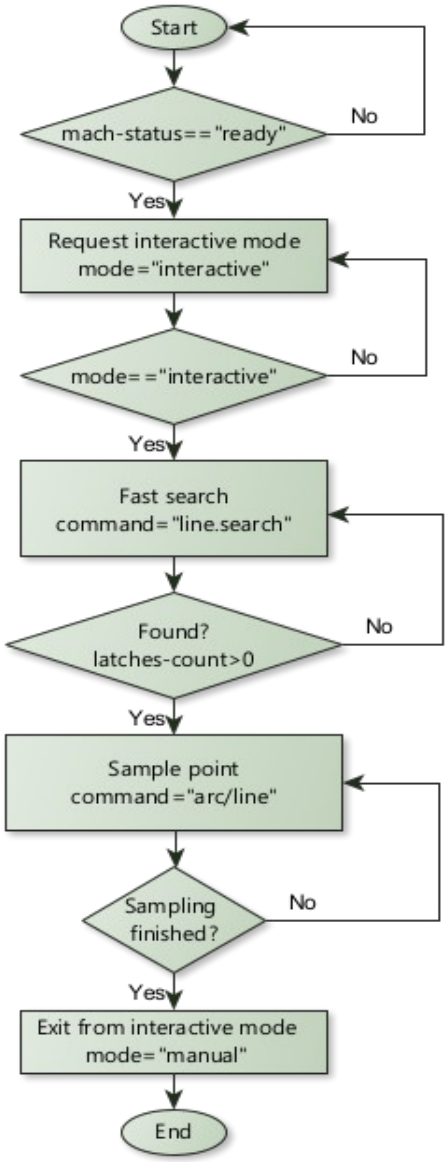
The `latches-count` field, always null, is included for symmetry with the other commands.

38 Don't forget the `latches-queue-max-size` limit: if the number of latches is equal to this number, there may have been additional ignored latches in the movement.

Example of a minimal session

```
⇨ [%]id=1620[%]mode="interactive";tool="probe"[%]
  ⇦ [%]rep-to=1620[%]
⇨ [%]id=1621[%]command="line.search";from=(0,200);to=(1000,200);edge-type=rise;
      speed-perc=100[%]
  ⇦ [%]rep-to=1621[%]
  ⇦ [%]msg="done 1621"[%]latches-count=1;latch-1=r(153.457,200);probe-status=1;
        x-tool-pos=384.843;y-tool-pos=200.001[%]
⇨ [%]id=1622[%]command="arc+";from=(120,200);to=(180,200);center=(150,200);
      edge-type=rise,fall;speed-perc=50[%]
  ⇦ [%]rep-to=1622[%]
  ⇦ [%]msg="done 1622"[%]latches-count=0;probe-status=0;x-tool-pos=179.998;y-tool-pos=200[%]
⇨ [%]id=1623[%]mode="manual"[%]
  ⇦ [%]rep-to=1623[%]
```

Flowchart



The search of the first point is usually performed at high speed and stopping on the first latch

Point sampling is typically performed at low speed with small movements

To exit, simply desynchronize

8.10 Float Scheme descriptor (mac)

The general format described in 4 “Scheme descriptor format (mac)” above on p.36 is specialized for *Float* machines through the [paths] section, in which are specified the tools and their respective *paths* for processing the *Scheme*.

A *path* is a sequence of entities (arcs and lines[39]) composing a working trajectory that can be processed with a certain tool.

For example, a cut like X 1000 is a *path* made by a single *entity*, a vertical line.

A sequence of paths using the same tool constitutes a work *stage*.

Typical *stages* include *TPF* removal, low-E deletion, rectilinear scoring, curvilinear scoring, marking.

The chronological sequence of these *stages* matters: for example low-E deletion shouldn’t be performed on a *path* that has already been scored.

8.10.1 [paths] section

It contains a sequence of *stages* that, grouping the *paths* by their tool, reflects the various steps of monolithic glass processing.

The format is similar[40] to the one used to describe generic shapes. In this context, each ini key entry represents a *stage* and consists of:

- Declaration of the tool(s)[41]
- List of associated *paths*[42]

Although the content of [paths] could, in principle, be created from the data already provided in the mac header, the possibility to explicitly define it allows for full control over the actual processing and impose:

- Translation of cuts (*Sheet trims*)
- Sequence and direction of cuts (movement optimization)
- Application of processing parameters configured by customers in the *HMI* (ex. minimum distance from edges, unidirectional cuts, etc.)
- Special processing (structural/multi-pass low-E deletion, defect marking, additional score lines to open contours, …)

Regarding the generation of the [paths] section, we will develop the following focal points:

- Declaration of generation parameters
- Paths notation
- Cuts direction
- Sequence of operations
- Level of detail
- Paths generation rules
- Technological aspects

39 For details refer to sect. 6.3.1 “Paths, entities, shapes” above on p.52
40 Actually, the same *parser* is used (see sect. 6.4.1 above on p.54), expanded to support additional notations
41 See sect. 6.4.2 “Path attributes declaration: tools” above on p.55
42 For the general format refer to table “Paths definition” above on p.54 in section 6.4.1

8.10.2 Declaration of generation parameters

The section can be opened with the optional `parameters` field, which allows to explicitly declare the criteria used to generate the *paths*.
This data provides information on how to interpret the generated work, thereby enabling consistency checks (either automated or by visual inspection) that are useful for identifying potential errors.

Field	Value	Description	Example
`size`	*double x double*	Overall glass dimensions considered for margins from glass edges	`size:1000x500`
`trims`	*(double, double)*	Trims (x/vertical, y/horizontal) applied to the *Sheet* (translation of cuts)	`trims:(20,20)`
`tools`	*string*	Tools explicitly considered in the generation **Tool** / **Description** `tpf` — *TPF* removal scraper `lowe` — low-E wheel `knife` — Vinyl/paper cutting blade `score` — Scoring head(s) `open` — Breakout tool(s) `mark` — Marker pen See also: `mach-capabilities`	`tools:lowe,score`
`tpf-width`	*double*	Width of the TPF scraper. Affects the edge margin of parallel cuts and multi-pass (structural) processing	`tpf-width:20`
`tpf-margin`	*double*	Minimum distance of the orthogonal paths from edges	`tpf-margin:10`
`lowe-width`	*double*	Width of the low-E wheel. Affects multi-pass (structural) processing	`lowe-width:20`
`lowe-margin`	*double*	Minimum distance of the orthogonal paths from edges	`lowe-margin:0`
`lowe-overlap`	*double*	Overlap of low-E wheel passes used in the case of multi-passes on a wider area (structural deletion)	`lowe-overlap:1.5`
`score-edges-margin`	*double*	Minimum distance of scores from glass edges	`score-edges-margin: 3`
`score-cross-margin`	*double*	Minimum distance of scores from intersections	`score-cross-margin: 2`

The units of measurement are declared in the `mac` header.

8.10.3 Paths notation

Rectilinear cuts

Typically, the primary work consists of making cuts parallel to the sides of the *Sheet*, dividing it into rectangular pieces. This is known as rectilinear[43] processing.

Considering a vertical cut at `X=1000` on a sheet `3210` high, it can be described in this general form:

```
score=path:{
        start:{from:(1000,0)}
        line:{to:(1000,3210)}
       }
```

This notation is perfectly valid and has the advantage of being generic.

Due to the frequency of these parallel cuts, a more concise alternative notation is introduced:

```
score=line:{ x:1000, y:0|3210 } ; comma between json fields is optional
```

If the *Sheet* were exactly `3210` high, this would be a *cross-cut*: both `y` ordinates[44] are on *Sheet* edges, and therefore, if the cut direction is toward positive `y`, can be omitted without ambiguity:

```
score=line:{ x:1000 } ; toward positive y
```

For example, a horizontal trim with height `20` across the entire *Sheet*:

```
score=line:{ y:20 } ; toward positive x
```

Be careful not to confuse the `x` and `y` coordinates with the `XYZW` cutting levels of the cut tree in this context!

Whether to use this concise notation is left to the implementer discretion: while it offers the benefit of reducing file size and improving readability for direct inspection, it also complicates the generation process.

The following notations are equivalent[45]:

```
*=line:{ x:900|0 y:500 }
```

```
*=path:{
    start:{from:(900,500)}
    line:{to:(0,500)}
   }
```

Grouping of paths in stages

Each tool entry in the *ini* file indicates a separate work *stage*[46]. When multiple consecutive *paths* are part of the same *stage* (ex. have the same tool), it's advisable to group them in the same entry, as shown in the following example:

```
score=line:{ x:1000 y:5|3205 }, ; comma required here!
      line:{ x:1000|5 y:500}
```

If the two *paths* were written with two separate entries:

```
score=line:{ x:1000 y:5|3205 }
score=line:{ x:1000|5 y:500}
```

Due to discrepancies between `ini` and `json`, it is necessary to explicitly include commas to separate the lists of root `json` nodes that span multiple lines (see sect. 11.2 “Extended json-ini syntax” below on p.154).

This would involve designating two separate *stages*. A typical example where this is beneficial is distinguishing between rectilinear and curvilinear scoring.

Grouping *paths* has a semantic significance, meaning they are combined into distinct work *stages*. The notations shown below are not equivalent, the one on the right implies two separate *stages*:

```
score=line:{x:900},
      line:{x:900|0 y:500}
```

```
score=line:{x:900}
score=line:{x:900|0 y:500}
```

43 Distinct from curvilinear processing, which involves generic paths for shapes. This distinction is important because their processing requires different scoring speeds/pressures and sometimes even dedicated scoring heads.

44 Referring to nominal targets (start/end points), without considering any tool margins.

45 Two notations are considered equivalent if they produce the same processing operations.

46 The machine may then split or combine the specified *stages* as necessary.

8.10.4 Cuts direction

The ability to specify the cuts direction allows for preferences such as unidirectional cuts or to impose the outcome of a movement optimization algorithm.

The following is a horizontal cut at a height of 30, executed towards the positive X direction:

```
score=path:{
        start:{from:(0,30)}
        line:{to:(6000,30)}
       }
```

Representable more concisely as:

```
score=line:{ x:0|6000 y:30 } ; positive direction
```

And if the *Sheet* were exactly 6000 wide:

```
score=line:{ y:30 } ; positive direction implied
```

The following is the same cut but executed in the opposite direction:

```
score=path:{
        start:{from:(6000,30)}
        line:{to:(0,30)}
       }
```

Representable more concisely as:

```
score=line:{ x:6000|0 y:30 } ; negative direction
```

8.10.5 Coordinates translation

You can set the coordinate reference origin at various levels.
A global origin can be defined and remains active until the next redefinition:

```
score=line:{ x:200|400 y:500 } ; cut from 200 to 400 at height 500
origin=(20,50)
score=line:{ x: 200|400 y:500 } ; cut from 220 to 420 at height 550
origin=(-10,5)
score=line:{ x:200|400 y:500 } ; cut from 190 to 390 at height 505
```

To the global origin are added possible local origins defined just for the *stage* and for the single *path*.
In the following example, the vertical cut will be made at X=30:

```
origin=(10,0) ; global origin
*=origin:(10,0), ; stage origin
  path:{
    start:{origin:(10,0) from:(0,0)} ; path origin
    line:{to:(0,500)}
   }
```

This notation allows the file creator to explicitly specify the cut coordinates, separating them from the translations due to *Trims* and the position of *Stripes*.
While this requires additional effort, it enhances readability for humans.

The following notations are equivalent:

```
origin(20,30)
*=line:{x:900 y:0|600}
```

```
*=line:{x:920 y:30|630}
```

```
origin(20,30)
*=path:{
    start:{from:(900,0)}
    line:{to:(900,600)}
   }
```

```
*=path:{
    start:{from:(920,30)}
    line:{to:(920,630)}
   }
```

8.10.6 *Stage* bounding box

For each entry of the `[paths]` section it's also possible to define a bounding rectangle, within which all the *paths* are supposed to be contained:

```
[paths]
*=box:1000x500,
  path:{ … }, …
  path:{ … }
```

> ⚠ The origin of `box` is equal to the current origin, so it should be defined after the local `origin` if present.

Declaring the bounding box allows for the automatic application of the correct tool margins and cropping the contained *paths*.

8.10.7 Referencing shape definitions

Specifying `origin` and `box` allows for the direct use of the `[shape]` definition in `[paths]`.

For instance, the processing of the following semicircle in a piece located at `(800;20)` is nearly identical to its original definition:

```
[shape]
name="semicircle"
size=500x250
*=path:{
    start:{from:(500,250)}
    arc-:{to:(0,250) C:(250,250)}
   }
```

```
[paths]
score=origin:(800,20), box:500x250,
  path:{
    start:{from:(500,250)}
    arc-:{to:(0,250) C:(250,250)}
   }
```

In the `[paths]` section, it is possible to reference the shape definition directly using the `name` field (which is always recommended for clarity).

The following notation is equivalent to the one above:

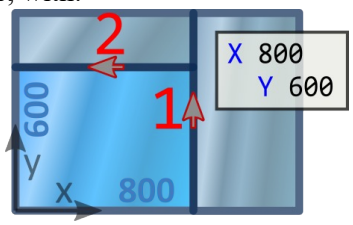
```
[paths]
score=name:"semicircle", origin:(800,20)
```

The `box` field is no longer necessary: if omitted, it will be considered equal to the `[shape]` definition `size`.

8.10.8 Sequence of operations

The sequence of cuts is determined by their definition order, for example, with:

```
score=line:{ x:800 },
      line:{ x:800|0 y:600 }
```



The machine will first perform a vertical incision (from bottom to top) across the entire height of the *Sheet*, obtaining a *Stripe* `800` wide, then it will perform a horizontal incision at a height of `600` on this *Stripe*, moving back toward zero.

When multiple tools are specified for a group of *paths*, the chronological sequence of the *stages* is determined by their priority: `tpf` always precedes `lowe`, which always precedes `score`[47].

For example, assuming all the tools share the same *paths*, the notation:

`tpf,lowe,score=`*`<common-paths>`*`...`

Is equivalent to:

`tpf=`*`<common-paths>`*`...`

`lowe=`*`<common-paths>`*`...`

`score=`*`<common-paths>`*`...`

Involving three *stages*: the TPF removal, then the low-E deletion, and finally the scoring.

> The following notation will also produce the same processing:
> `score,lowe,tpf=<common-paths>...`
> But it is preferable to avoid it because it might confuse a human reader.

When the tools/*stages* do not share the same *paths*, they must be specified separately:

`tpf=`*`<tpf-paths>`*`...`

`lowe=`*`<lowe-paths>`*`...`

`score=`*`<score-paths>`*`...`

Note that in this case, it is crucial to pay attention to the order of definition, which will be strictly followed in the generation of the work.

The following notation will result in problematic processing (low-E wheel clogged in the *TPF*):

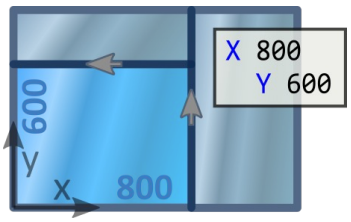
`lowe=`*`<lowe-paths>`*`...`

`tpf=`*`<tpf-paths>`*`...`

`score=`*`<score-paths>`*`...`

47 Tools not related to the material type, such as the marker pen (`mark`), are excluded from these considerations. Their *paths* must always be specified separately.

8.10.9 Level of detail



It is possible to define the `[paths]` section with different levels of detail. Consider the simple *Job* illustrated in the adjacent figure, consisting of two cuts.

Level **zero** consists of omitting the `[paths]` section entirely, allowing the machine to generate it automatically.

Level **one** involves indicating only the sequence and direction of the cuts, in this example:

```
*=line:{ x:800 },
  line:{ x:800|0 y:600 }
```

This notation enables imposing movements optimization without the need to specify additional details; the machine will automatically select the appropriate tools based on the material[48] and apply the correct margins for each tool.

> For levels zero and one is crucial to declare `glass-type` to determine the necessary tools (ex. `glass-type=tpf,lowe`).

Level **two** involves explicitly detailing the various work *stages* (tools sequence) without worrying about margins or other constraints. Assuming a low-E material, the example becomes:

```
lowe=line:{ x:800 },
  line:{ x:800|0 y:600 }
score=line:{ x:800 },
  line:{ x:800|0 y:600 }
```

> The actual processing coordinates will likely differ from those indicated (nominal) because the machine will apply appropriate margins based on the tool.

Level **three** specifies the *paths* applying tool constraints (margins from edges and intersections). Assuming a *Sheet* with height `900`, the example becomes:

```
lowe=line:{ x:800 y:5|895 },
     line:{ x:800|5 y:600 }
score=line:{ x:800 y:2|898 },
      line:{ x:798|2 y:600 }
```

Where a margin of `5 mm` for the low-E wheel and `2 mm` for the scoring head are enforced[49].

This level of detail provides complete control[50], allowing management of tool constraints, imposing a sequence, and describing special processing (structural low-E deletion, …).

> Currently, automatic job generation is not mature: no movements optimization, margins from intersections are not ready, no trajectory offset algorithms for multiple passes.

> Initially, adopting the most detailed descriptor will be natural to maintain backward compatibility with *HMI* settings and manage the transition gradually.

The long-term goal is to gradually reduce the level of detail to simplify the *HMI* by removing machine-specific details.

> It's generally advisable not to mix different levels of detail. For example, the following notation has an unclear intent:
>
> `*=<path1>…`
>
> `some-tool=<path2>…`
>
> `*=<path3>…`
>
> The execution of the `<path2>` *stage* in relation to `<path1>` and `<path3>` multiple *stages* is ambiguous.

48 The asterisk means “select automatically the tools”.

49 The machine will still apply its own constraints if they are more stringent.

50 Clearly, this requires regenerating the `mac` file whenever the parameters that influence its content change.

8.10.10 Paths generation rules

1) The *paths* must be within the glass area, so all coordinates are positive and less than a certain dimension
2) The starting point of a *path* never coincides with the endpoint of the preceding *path* in the same *stage*. In such cases, the new *entity* is simply appended to the previous *path*

```
lowe=path:{
    start:{from:(0,500)}
    line:{to:(500,500)}
   }, path:{
    start:{from:(500,500)}
    line:{to:(600,800)}
   }
```

➡

```
lowe=path:{
    start:{from:(0,500)}
    line:{to:(500,500)}
    line:{to:(600,800)}
   }}
```

ℹ A new *path* is introduced only when it is necessary to move to another point (without tool) or if the tool changes.

3) When specifying the *paths* for each tool, it's crucial to pay attention to the order of definition, as it will be followed precisely in the work sequence.

```
score=line:{x:500 y:10|500}
lowe=line:{x:500 y:10|500}
tpf=line:{x:500 y:10|500}
```

➡

```
tpf=line:{x:500 y:10|500}
lowe=line:{x:500 y:10|500}
score=line:{x:500 y:10|500}
```

ℹ Defining a scoring *stage* before the low-E deletion is probably not a good idea.

4) *Paths* declared with the same tool should be grouped under the same *stage* unless there is a desire to highlight two logically distinct operations

```
lowe=line:{x:500 y:10|500}
lowe=line:{x:500|10 y:500}
```

➡

```
lowe=line:{x:500 y:10|500},
     line:{x:500|10 y:500}
```

ℹ *Paths* shouldn't be divided without a good reason, such as distinguishing between rectilinear and curvilinear processing or specifying a certain shape.

5) Separate the rectilinear and curvilinear scores into two distinct *stages*

```
score=line:{ y:0 },
      line:{ x:0 y:0|1180 },
    path:{
    start:{from:(900,500)}
    arc-:{to:(100,500) C:(500,500)}
   }
```

➡

```
score=line:{ y:0 },
  line:{ x:0 y:0|1180 }
score=path:{
    start:{from:(900,500)}
    arc-:{to:(100,500) C:(500,500)}
   }
```

ℹ Rectilinear and curvilinear scores require different strategies and therefore must be separated.

6) Rectilinear scoring *stage* should precede the curvilinear one

```
score=path:{
    start:{from:(900,500)}
    arc-:{to:(100,500) C:(500,500)}
   }
score=line:{ y:0 },
      line:{ x:0 y:0|1180 }
```

➡

```
score=line:{ y:0 },
      line:{ x:0 y:0|1180 }
score=path:{
    start:{from:(900,500)}
    arc-:{to:(100,500) C:(500,500)}
   }
```

ℹ It is better to score the *Primitives* first to contain possible inner cracks avoiding their propagation.

7) Avoid mixing different levels of detail

```
*=line:{y:10}
lowe=line:{y:20}
*=line:{y:30}
```

➡

```
lowe=line:{y:10},
  line:{y:20},
  line:{y:30}
score=line:{y:10},
  line:{y:30}
```

ℹ Specifying tools for only some paths would result in ambiguous notations.

8.10.11 Technological aspects

Scoring

The glass is cut by scoring it (scratching its surface) and then literally breaking (opening) the score.

The scoring is done by a *head* equipped with a carbide wheel, oriented tangentially to the cut trajectory.

The scoring of rectilinear and curvilinear cuts is treated differently:

- Scoring a curve requires slower speed and more pushing force
- To maximize precision, horizontal scores are always done with the scoring head oriented at 0° and vertical ones with the scoring head oriented at 90°, regardless of the score direction

In the past there have been cases where the customer requested to force the direction of the scores (unidirectional scoring).

The scoring head should never go outside the glass, each score should start or end at a safe distance from the glass edges.

The scores should never cross each other.

Here are the invariants related to this process:

- Both the starting point and the endpoint of a score are always near another perpendicular score or near a glass edge

low-E deletion

The cut of low-E glass involves an initial phase where the low-E coating is deleted with a wheel, followed by the standard scoring process.

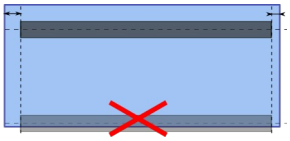
The low-E wheel tool is characterized by a width (`lowe-width`) and may go partially outside the glass; the dust created by the operation is vacuumed through a tube placed near the tool.

Here are the invariants related to this process:

- The low-E deletion *stage* always precedes the score
- The scores are always contained within the deleted low-E area

TPF scraping

The cut of *TPF*-protected glass involves an initial phase where the protective film is removed from the cuts with a scraper, followed by processing as a standard low-E material.



The *TPF* scraper is also characterized by a width (`tpf-width`) and should never go outside the glass edges, even partially (`tpf-margin`).

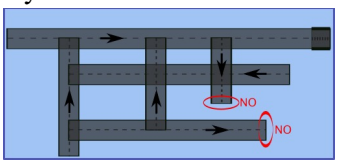
Here are the invariants related to this process:

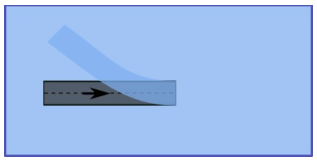
- The *TPF* removal *stage* always precedes the low-E deletion
- The low-E deletion area is always contained within the removed *TPF* area

> ⚠ This last invariant is not automatic, as the two tools have different constraints. In the case of *TPF* processing, it is therefore necessary to ensure that the margins from the edges for the low-E wheel are aligned with those adopted for the *TPF* scraper.

There are some details handled internally by the machine:

Before going up in an area not already processed, the scraper must cut the plastic strip, otherwise it would remain attached to the glass.

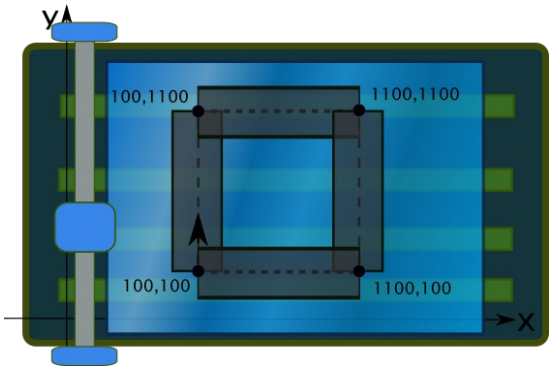




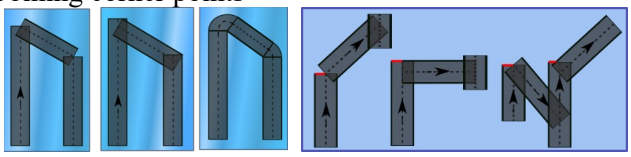
The removed plastic strip must be regularly cut to facilitate disposal by the vacuum system.

Handling tools size: joining trajectories

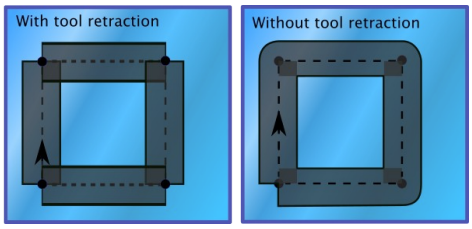
Tools can have a non-negligible size that must be taken into account.



- Joining corner points



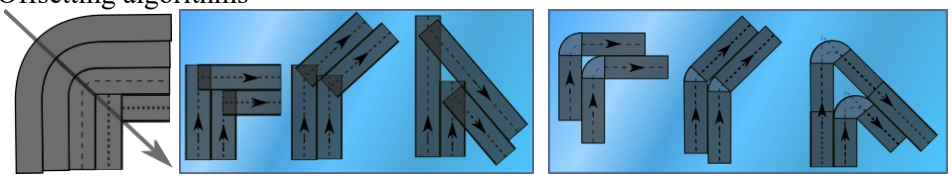
- Effect of tool retraction



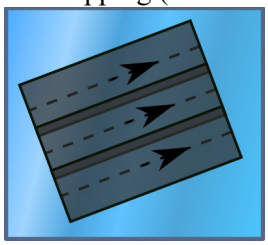
Handling tools size: processing areas

If you have to process an area wider than the tool width, you'll have to perform multiple passes offsetting the original trajectory.

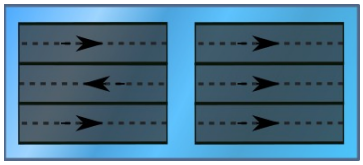
- Offsetting algorithms



- Overlapping (see `lowe-overlap`)



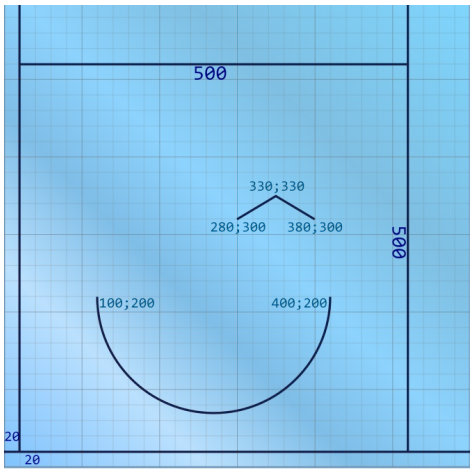
- Force uniform direction if desired



8.10.12 Describing a simple multi-tool Job

Let's consider a relatively simple example to show the [paths] section when the processing involves with multiple tools with a *TPF* glass, where for each cut is necessary to remove the film and the low-E before the scoring. The first part of the file is common to all machines (for details see sect. 4 "Scheme descriptor format (mac)" above on p.36):

```
[project]
name="float-smile"
scheme=1/1

[scheme]
type="float"
size=600x600x3
glass-id="FL3_TPF"
glass-type=tpf,lowe
trims=(20,20)
X=500
  Y=500 id:1

[pieces]
1={ size:500x500 shape:"smile" }

[shape]
name="smile"
size=500x500
*=path:{
    start:{from:(100,200)}
    arc+:{to:(400,200) C:(250,200)}
   },
  path:{
    start:{from:(380,300)}
    line:{to:(330,330)}
    line:{to:(280,300)}
   }
```



It's possible to specify some details about the material, such as thickness (the third dimension in `size`, ex. `size=600x600x3`) and the required processing (ex. `glass-type=tpf,lowe`).

Both information are optional but highly recommended for *Float* processing: specifying thickness is especially important in case of a *Line* equipped with a *Stripe* breakout automatic system, and the `glass-type` is crucial if [path] section is absent or explicit with minimal detail.

Paths (minimal detail)

```
[paths]
parameters = {
    size: 600x600
    trims: (20,20)
   }

origin=(20,20)

*=line:{ y:0 },
  line:{ x:0 y:0|580 },
  line:{ x:500 y:0|580 },
  line:{ x:500|0 y:500 }

*=name:"smile", origin:(0,0),
box:500x500,
  path:{
    start:{from:(100,200)}
    arc+:{to:(400,200) C:(250,200)}
   },
  path:{
    start:{from:(380,300)}
    line:{to:(330,330)}
    line:{to:(280,300)}
   }
```

Here's an example of a *Job* defined with minimal detail. This type of description provides only theoretical paths and lets the machine autonomously select tools and margins.

Enabling this output is currently optional; note that in this case it's important to specify the necessary tools in the `glass-type` field under the [scheme] section.

In this case, since the *Shape*'s paths coincide with its definition, it would have been possible to simply write:

```
*=name:"smile", origin:(0,0)
```

to reuse the definition specified above.

Paths (maximum detail)

Now let's specify the *Job* description for the *TPF* material, ensuring the application of certain tool margins.

```
[paths]
parameters = {
    size: 600x600
    trims: (20,20)
    tools: "tpf,lowe,score"
    tpf-width: 20
    tpf-margin: 5
    lowe-width: 20
    lowe-margin: 5
    lowe-overlap: 2.5
    score-edges-margin: 3
    score-cross-margin: 2
   }

tpf=line:{ x:5|595 y:20 },
  line:{ x:20 y:20|595 },
  line:{ x:520 y:20|595 },
  line:{ x:520|20 y:520 }

tpf=origin:(20,20), ; smile
  path:{
    start:{from:(100,200)}
    arc+:{to:(400,200) C:
(250,200)}
   },
  path:{
    start:{from:(380,300)}
    line:{to:(330,330)}
    line:{to:(280,300)}
   }

lowe=line:{ x:5|595 y:20 },
  line:{ x:20 y:20|595 },
  line:{ x:520 y:20|595 },
  line:{ x:520|20 y:520 }

lowe=origin:(20,20), ; smile
  path:{
    start:{from:(100,200)}
    arc+:{to:(400,200) C:
(250,200)}
   },
  path:{
    start:{from:(380,300)}
    line:{to:(330,330)}
    line:{to:(280,300)}
   }

score=line:{ x:3|597 y:20 },
  line:{ x:20 y:22|597 },
  line:{ x:520 y:22|597 },
  line:{ x:518|22 y:520 }

score=origin:(20,20), ; smile
  path:{
    start:{from:(100,200)}
    arc+:{to:(400,200) C:
(250,200)}
   },
  path:{
    start:{from:(380,300)}
    line:{to:(330,330)}
    line:{to:(280,300)}
   }
```

The `parameters` field is entirely optional and serves only to declare the generation criteria: translation due to trims, considered tools and their characteristics.

It is advisable to include it to correctly interpret the generated processing, facilitating debugging.

For details see sect. 8.10.2 “Declaration of generation parameters” above on p.122.

The definition sequence will be strictly followed in the processing: defining a `lowe` *stage* before `tpf` would create significant issues.

In this example, the processing of straight cuts is separated from that of *Shape*s also for `tpf` and `lowe`: this isn’t strictly necessary but was done for symmetry with `score`.

The following more general notation is allowed:

```
[paths]
tpf=path:{
    start:{from:(5,20)}
    line:{to:(595,20)}
   },
  path:{
    start:{from:(20,20)}
    line:{to:(20,595)}
   },
  path:{
    start:{from:(520,20)}
    line:{to:(520,595)}
   },
  path:{
    start:{from:(520,520)}
    line:{to:(20,520)}
   },
  path:{
    start:{from:(120,220)}
    arc+:{to:(420,220) C:(270,220)}
   },
  path:{
    start:{from:(400,320)}
    line:{to:(350,350)}
    line:{to:(300,320)}
   }

lowe=path:{
    start:{from:(5,20)}
    line:{to:(595,20)}
   },
  path:{
    start:{from:(20,20)}
    line:{to:(20,595)}
   },
  path:{
    start:{from:(520,20)}
    line:{to:(520,595)}
   },
  path:{
    start:{from:(520,520)}
    line:{to:(20,520)}
   },
  path:{
    start:{from:(120,220)}
    arc+:{to:(420,220) C:(270,220)}
   },
  path:{
    start:{from:(400,320)}
    line:{to:(350,350)}
    line:{to:(300,320)}
   }

score=path:{
    start:{from:(3,20)}
    line:{to:(597,20)}
   },
  path:{
    start:{from:(20,22)}
    line:{to:(20,597)}
   },
  path:{
    start:{from:(520,22)}
    line:{to:(520,597)}
   },
  path:{
    start:{from:(518,520)}
    line:{to:(22,520)}
   }

score=path:{
    start:{from:(120,220)}
    arc+:{to:(420,220) C:(270,220)}
   },
  path:{
    start:{from:(400,320)}
    line:{to:(350,350)}
    line:{to:(300,320)}
   }
```

This notation requires less generation effort because its movements map 1:1 with G-code commands.

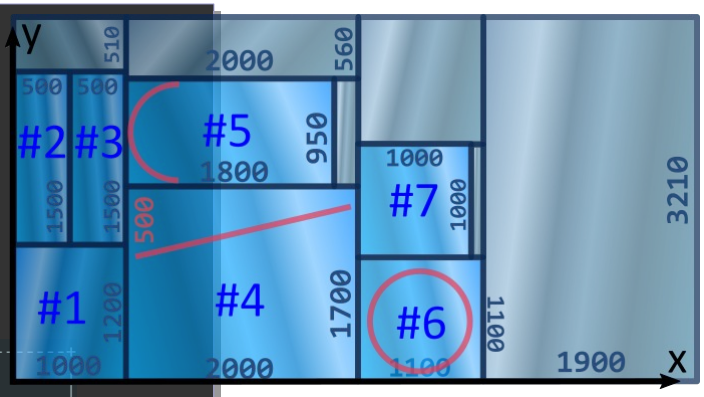
Note how the *Shapes* are always processed after the rectilinear cuts: this is desirable for scoring to prevent any breaks from spreading outside the *Primitive*.

This criterion extends to other tools to ensure predictable processing (*least surprise*).

8.10.13 Describing a typical *Scheme*

Let's take a look at a more realistic schema. The common declarations don't present any surprises (see sect. 4 “Scheme descriptor format (mac)” above on p.36):

```
[project]
name="float-first-demo"
scheme=1/1

[options]
units = { length: 0.001 } ; [mm]
granularity = { length: 0.1 }

[scheme]
type="float"
size=6000x3210x4
glass-id="FL4LE"
glass-type=lowe
trims=(20,20)
X=1000          ;                    6000
  Y=1200 id:1   ;      +-----+---------+-----+---------+
  Y=1500        ;      +--+--+---------++  // |
    Z=500 id:2  ;      |  |  |#5   950|+----++         |
    Z=500 id:3  ; 1500|#2|#3+---------++ #7 ||    //   | 3210
X=2000          ;     |--+--| #4      |----++         |
  Y=1700 id:4   ; 1200|     |     1700| #6  |         |
  Y=950         ;     | #1  |         | 1100|         |
    Z=1800 id:5 ;     +-----+---------+-----+---------+ -> X
X=1100          ;      1000    2000     1100    1900
  Y=1100 id:6
  Y=1000
    Z=1000 id:7

[pieces]
1={ size:1000x1200 descr:"first" label:0 lowe:"10,40,80,60" }
2={ size:500x1500 descr:"second" }
3={ size:500x1500 descr:"third" }
4={ size:2000x1700 descr:"fourth" shape:"trapezoid" }
5={ size:1800x950 descr:"fifth" shape:"arch" }
6={ size:1100x1100 descr:"sixth" shape:"circle" }
7={ size:1000x1000 descr:"seventh" }

[labels]
label-size=100x80
alignment=C

[shape]
name="trapezoid"
size=2000x1700
*=path:{
    start:{from:(2000,1600)}
    line:{to:(0,1500)}
   }

[shape]
name="arch"
size=1800x950
*=path:{
    start:{from:(475,950)}
    arc+:{to:(475,0) C:(475,475)}
   }

[shape]
name="circle"
size=1100x1100
*=path:{
    start:{from:(50,550)}
    line:{to:(1050,550) bulge:1}
    line:{to:(50,550) bulge:1}
   }
```



Since only scoring is required, the `glass-type` field has been omitted rather than left blank.

Paths (maximum detail)

Here's an example of the [paths] section with maximum detail and generic notation.

```
[paths]
parameters = {
    size: 6000x3210
    trims: (20,20)
    tools: "score"
    score-edges-margin: 3
    score-cross-margin: 2
   }

score=path:{
    start:{from:(4120,3)}
    line:{to:(4120,3207)}
   },
  path:{
    start:{from:(4118,20)}
    line:{to:(3,20)}
   },
  path:{
    start:{from:(1020,22)}
    line:{to:(1020,3207)}
   },
  path:{
    start:{from:(1018,2720)}
    line:{to:(3,2720)}
   },
  path:{
    start:{from:(20,2718)}
    line:{to:(20,22)}
   },
  path:{
    start:{from:(22,1220)}
    line:{to:(1018,1220)}
   },
  path:{
    start:{from:(520,1222)}
    line:{to:(520,2718)}
   },
  path:{
    start:{from:(3020,22)}
    line:{to:(3020,3207)}
   },
  path:{
    start:{from:(3018,2670)}
    line:{to:(1022,2670)}
   },
  path:{
    start:{from:(1022,1720)}
    line:{to:(3018,1720)}
   },
  path:{
    start:{from:(2820,1722)}
    line:{to:(2820,2668)}
   },
  path:{
    start:{from:(3022,2120)}
    line:{to:(4118,2120)}
   },
  path:{
    start:{from:(4118,1120)}
    line:{to:(3022,1120)}
   },
  path:{
    start:{from:(4020,1122)}
    line:{to:(4020,2118)}
   }

score=path:{
    start:{from:(3070,570)}
    arc+:{to:(4070,570) C:(3570,570)}
    arc+:{to:(3070,570) C:(3570,570)}
   },
  path:{
    start:{from:(3018,1619.9)}
    line:{to:(1022,1520.1)}
   },
  path:{
    start:{from:(1451.457,2668)}
    arc+:{to:(1022,2238.543) C:(1495,2195)}
   },
  path:{
    start:{from:(1022,2151.457)}
    arc+:{to:(1451.457,1722) C:(1495,2195)}
   }
```

This example illustrates the case of scoring only. For other tools, the process wouldn't differ much, except for the length. For an example of multi-tool processing, refer to the previous example.

Note how the cut targets already account for the necessary margins from other crossing cuts and from the glass edges. The *Shapes* have also been appropriately cropped to prevent overlaps with the sides of their *Primitives*.

Paths (minimal detail)

Now let's look at a [paths] section specified using minimal detail and a readability-oriented notation: tools not specified, nominal paths, concise notation for straight cuts, and the use of origin shifts to directly use the pieces dimensions:

```
[paths]
parameters = {
    size: 6000x3210
    trims: (20,20)
   }

origin=(20,20) ; Translation due to Trims

; Rectangles
*=line:{ x:4100 }, ; Subsheet
  line:{ x:4100|-20 y:0 }, ; Horizontal Trim
  origin:(0,0), ; First Stripe
  line:{ x:1000 y:0|3190 },
  line:{ x:1000|-20 y:2700 },
  line:{ x:0 y:2700|0 }, ; Vertical Trim
  line:{ x:0|1000 y:1200 }, ; Piece 1
  line:{ x:500 y:1200|2700 }, ; Pieces 2 and 3
  origin:(1000,0), ; Second Stripe
  line:{ x:2000 y:0|3190 },
  line:{ x:2000|0 y:2650 },
  line:{ x:0|2000 y:1700 }, ; Piece 4
  line:{ x:1800 y:1700|2650 }, ; Piece 5
  origin:(3000,0), ; Third Stripe
  line:{ x:0|1100 y:2100 },
  line:{ x:1100|0 y:1100 }, ; Piece 6
  line:{ x:1000 y:1100|2100 } ; Piece 7

; Shape of piece 6
*=name:"circle", origin:(3000,0),
box:1100x1100,
  path:{
    start:{from:(50,550)}
    arc+:{to:(1050,550) C:(550,550)}
    arc+:{to:(50,550) C:(550,550)}
   }
; Shape of piece 4
*=name:"trapezoid", origin:(1000,0),
box:2000x1700,
  path:{
    start:{from:(2000,1600)}
    line:{to:(0,1500)}
   }
; Shape of piece 5
*=name:"arch", origin:(1000,1700),
box:1800x950,
  path:{
    start:{from:(475,950)}
    arc+:{to:(475,0) C:(475,475)}
   }
```

Setting `origin` allows for explicitly referencing the cut dimensions visible in the *Scheme* declaration, making it easier to read.

The shapes processing are declared separately to define their respective bounding rectangle (`box`), which helps the G-code generator to properly crop them.

Naming the shape isn't mandatory but recommended: it facilitates inspection and allows referencing existing definitions. In this case, since the shape paths match their definitions, they could have been simply written as:

```
*=name:"circle", origin:(3000,0)
*=name:"trapezoid", origin:(1000,0)
*=name:"arch", origin:(1000,1700)
```

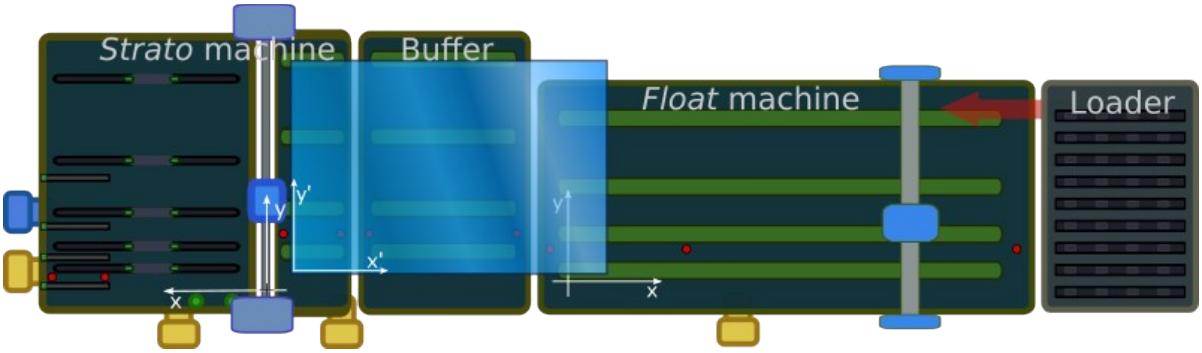
9 Combined processing

9.1 Objective

Examine the available interactions of *Strato* and *Float* machines combined in the same *Line*.

9.2 Context

Among MACOTEC offerings are combined *Lines*, consisting of a *Strato* machine with *Float* machine in place of the *Feed* table.

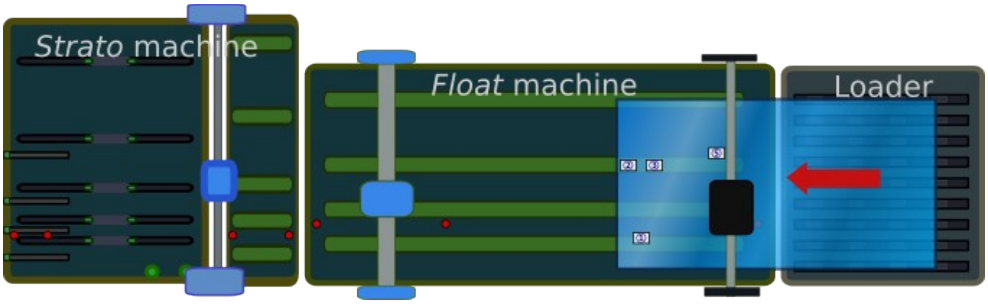


For the basic concepts see sect. 2.7 “Combined Lines” above on p.16.

9.3 *Labeling* Pieces of a *Float Scheme* on *Strato* machine

9.3.1 Scenario

Strato machines can operate a printer bridge[51] positioned just after the loading system to apply labels on the *Sheet* as it is transferred onto the *Feed* table. The prerequisites for the operation is to *synchronize* the desired *Scheme* on *Strato* machine, ensuring the correct labeling data is available.



On a combined *Line* equipped with a printer bridge, customers expects to label the *Pieces* of a *Float Project* as well: this requires that whenever a *Scheme* is *synchronized* on the *Float* machine, a corresponding `mac` file with the labeling data must also be *synchronized* on the *Strato* machine.

51 For further details refer to the full specification: SP061 – Ponte stampa su laminato.pdf

9.3.2 Example of labeling data

Here is an example of a `mac` file for a *Strato* machine containing just the labeling data of a *Scheme*.


```
[project]
name="LabelingDemo"
scheme=1/1

[scheme]
type="labels"
glass-id="FL4"
size=6000x3210
sheets=1
X=1000
  Y=1200 id:1
  Y=1500
    Z=500 id:2
    Z=500 id:3
X=2000
  Y=1700 id:4
  Y=950
    Z=1800 id:5
X=1100
  Y=1100 id:6

[pieces]
1={ size:1000x1200 customer:"EE" order:"FC2" descr:"first"  rack:"1" note1:"HH" }
2={ size:500x1500  customer:"EE" order:"FC2" descr:"second" rack:"2" note1:"MM" }
3={ size:500x1500  customer:"AM" order:"FC2" descr:"third"  rack:"2" note1:"MM" label:0 }
4={ size:2000x1700 customer:"AM" order:"FC2" descr:"fourth" rack:"1" note1:"HH" }
5={ size:1800x950  customer:"AM" order:"FC2" descr:"fifth"  rack:"1" note1:"HH" work:"TBB" }
6={ size:1100x1100 customer:"AM" order:"FC2" descr:"sixth"  rack:"1" note1:"HH" work:"TBB" }
                   ; +---------+
[labels]           ; | TL T TR |
label-size=100x80  ; | L  C  R |
alignment=C        ; | BL B BR |
                   ; +---------+
1={ pos:(500,600) print:"C:\MacoService\Labels\Label1.xml" }
2={ pos:(250,1950) print:"C:\MacoService\Labels\Label2.xml" }
3={ pos:(750,1950) piece:0 }
4={ pos:(2000,850) print:"C:\MacoService\Labels\Label4.xml" }
5={ pos:(1900,2125) print:"C:\MacoService\Labels\Label5.xml" }
6={ pos:(3550,550) print:"C:\MacoService\Labels\Label6.xml" }

; May be omitted (ex. just labeling a Float Scheme)
;[steps]
; ...
```

The `[steps]` section is omitted if there's no further processing to do.
Of course to load the data is still needed a synchronization on the *Strato* machine:

```
[%]id=1538[%]prj-name="LabelingDemo";prj-path="c:\SomeFolder"[%]
[%]id=1539[%]mode="auto";scheme=1[%]
```

The important part is the `[pieces]` section: the machine will try to apply a label for each *Piece*.

To avoid labeling a *Piece*, specify `label:0` or simply omit its entry entirely.

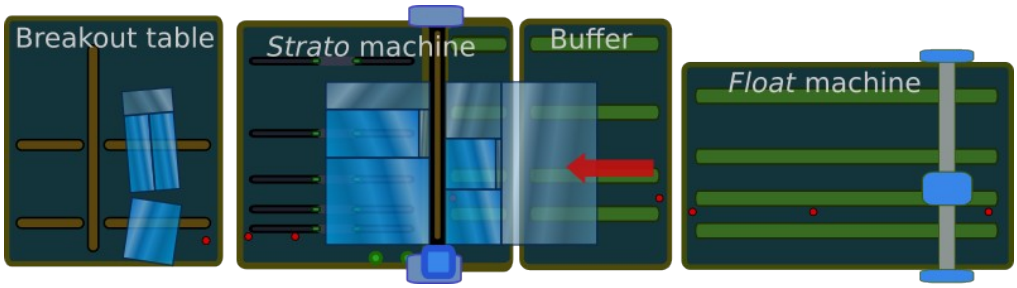
The fields of `[pieces]` entries may be used to generate the labels.

The `[labels]` section is optional and can be used to force the placement coordinates, otherwise the machine will calculate them using `label-size`, `alignment` and the cuts tree in `[scheme]`.
About the content of [labels] section, see sect. 4.5.6 above on p.40.

9.4 Automatic *Float Stripes* breakout on *Strato* machine

9.4.1 Scenario

The *Strato* machine can function as an automatic breakout system for the *Stripes* of a scored *Sheet* on *Float* machine.



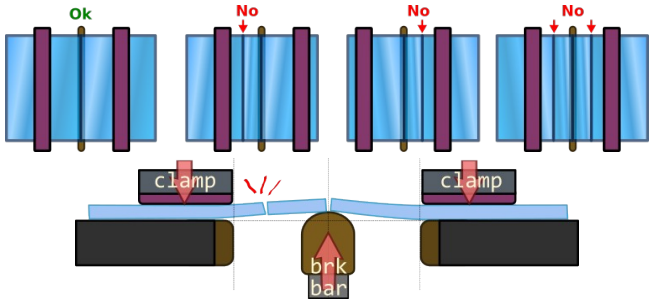
After scoring a *Scheme* on *Float* machine, the operator can command the *Strato* machine to automatically open the *Stripes* and deliver them downstream on the external breakout table.

9.4.2 Creating the *Stripes* breakout *Job*

When a *Scheme* is synchronized on the *Float* machine, a corresponding breakout *Job* is synchronized on the *Strato* machine.

Here are the basic directions on how to generate the *Job* description for *Stripes* breakout on *Strato*:

- [`scheme`] section
  - The material should be *Float*
  - The `type` field is set as `"stripes"`
- [`steps`] section
  - If specified, each row represents the breakout of a *Stripe*. As always, the machine cannot make 'intelligent' decisions to alter the *Steps*, as this would cause the *HMI* visualization to become misaligned with the machine's operations, so is responsibility of the *HMI* to decide which *Stripes* should be open. The criteria are:
    - *Product* and *Remnant* wider of a certain minimum threshold[52]
    - Avoid *Stripes* containing long scored lines parallel and near[52] the breakout bar



  - There's just one initial extraction from *Feed* (`extract`/`pop`)
  - No insertions (`insert`/`push`) or rotations (`rotate`)
  - The glass processing (`op`) is always `"open,detach"`
  - The *Product* is always moved out (`out-fwd`)
  - The last *Remnant* is moved out (`out-fwd`)

52 Related to the distance from the breakout bar and clamps, is chosen by the operator. A safe default value is `300 mm`.

9.4.3 Example of *Stripes* breakout *Job*

```
[project]
name="DemoFloatStripes"
scheme=1/2

[scheme]
type="stripes"
glass-id="FL4"
size=6000x3210
sheets=1
X=1000            ;                      6000
  Y=1200 id:1     ;       +-----+---------+-------++------+
  Y=900           ;       |     |---------|       ||      |
    Z=500 id:2    ;       |--+--| #5   950|   //  ||      |
    Z=500 id:3    ;   900|#3|#2|---------|       ||  //  | 3210
X=2000            ;       |--+--| #4      |-------||      |
  Y=1700 id:4     ;  1200|     |     1700| #6    ||      |
  Y=950 id:5      ;       | #1  |         |   1100||      |
X=1600            ;       +-----+---------+-------++------+ -->
  Y=1100 id:6     ;        1000    2000     1600  200
X=200
  Y=2000, id:7

[steps]
1="X 1000", status:{stack:1 proc:0}, sheet:{size:6000x3210 pop align:1000}, op:{open,detach},
  prod:{size:1000x3210 out-fwd}, remn:{size:5000x3210}
2="X 2000", status:{stack:0 proc:1}, sheet:{size:5000x3210 align:2000}, op:{open,detach},
  prod:{size:2000x3210 out-fwd}, remn:{size:3000x3210}
3="X 1600", status:{stack:0 proc:1}, sheet:{size:3000x3210 align:1600}, op:{open,detach},
  prod:{size:1600x3210 out-fwd}, remn:{size:1400x3210 out-fwd}
```

In case of no openable *Stripes*:

```
[project]
name="DemoFloatStripes"
scheme=2/2
                  ;             5900        100
[scheme]          ;     +-------------------+-+
type="stripes"    ;     |                   | |
glass-id="FL4"    ; <== | 3210              | |
size=6000x3210    ;     |          #1       | |
sheets=1          ;     |                   | |
X=5900 id:1       ;     +-------------------+-+
                  ;     +----> X
[steps]
1="OUT", status:{stack:1 proc:0}, sheet:{size:6000x3210 pop out-fwd}
```

Of course the labeling data may be added as described in the previous section to enable also *Piece* labeling.

The used material parameters are:

- `h-glass` [mm]
- `h-brkbar` [0:auto 1:normal 2:high]
- `t-brkbar` [s]
- `f-clamps-brk` [N/m]
- `f-clamps-max` [N]

For example: `sheet-type="float"; h-glass=4; wheel-angle=155; f-score-inf=11; f-score-sup=11; v-score=100; f-lowe=13; v-lowe=80; h-brkbar=0; t-brkbar=1; f-clamps-brk=1000`

9.5 Low-E deletion of a *Strato Scheme* on *Float* machine

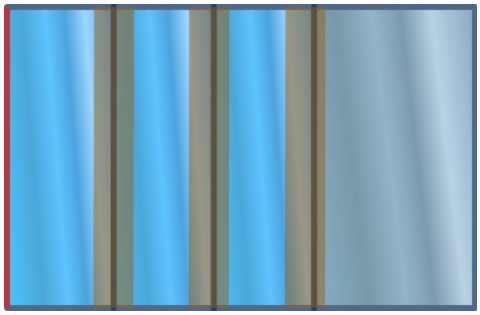
9.5.1 Context

Although the *Strato* machine can be equipped with a low-E wheel, performing the low-E deletion on a combined *Float* machine offers several advantages:

- Speed-up the processing on the *Strato* machine
- Perform structural low-E deletion (multiple passes for areas wider than the wheel)
- Perform deletion on *Shapes*

The misalignment problem

The primary challenge is addressing the misalignment between the low-E deletions performed by the *Float* machine and the cuts made by the *Strato* machine, in which errors tend to accumulate for each sibling cut[53].



The phenomenon is shown in the adjacent figure: while the low-E deletions are all executed in a single session referring to the reference side of the glass (in red), on the *Strato* machine the cuts are performed incrementally, referring progressively to different sides as the glass is cut.

This different references may lead to a misalignment between the low-E deletions and the cuts because in the latter operation errors tend to accumulate over time.

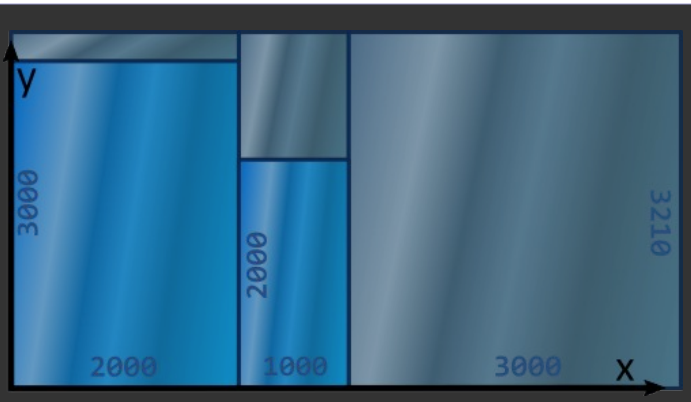
In the figure the *Strato* machine is cutting a little too much, possibly due to worn-out alignment blocks. As the machine continues to cut, the misalignment becomes progressively worse.

Due to this issue, the naive approach of performing the low-E deletion on the entire *Scheme* in one go on the *Float* machine is generally not acceptable.

9.5.2 Full deletion on *Strato*

As a comparison, let's consider an exemplificative basic low-E *Scheme Job* for *Strato* machines:

```
[project]
name="full-deletion-on-strato"
scheme=1/1

[scheme]
type="strato"
size=6000x3210
glass-id="33.1LE"
glass-type=lowe
X=2000
  Y=3000 id:1
X=1000
  Y=2000 id:2

[pieces]
1={ size:2000x3000 }
2={ size:1000x2000 }

[steps]
1="head-lowe", status:{stack:1 proc:0}, sheet:{size:6000x3210, extract align:0}, op:{lowe}
2="X 2000", status:{stack:0 proc:1}, sheet:{size:6000x3210 align:2000}, op:{full-cut}, prod:{size:2000x3210 rotate:-90}, remn:{size:4000x3210 insert}
3="head-lowe", status:{stack:1 proc:1}, sheet:{size:3210x2000 align:0}, op:{lowe}
4="Y 3000", status:{stack:1 proc:1}, sheet:{size:3210x2000 align:3000}, op:{full-cut}, prod:{size:3000x2000 piece:1}, remn:{size:210x2000 scrap}
5="X 1000", status:{stack:1 proc:0}, sheet:{size:4000x3210 extract align:1000}, op:{full-cut}, prod:{size:1000x3210 rotate:-90}, remn:{size:3000x3210 scrap}
6="head-lowe", status:{stack:0 proc:1}, sheet:{size:3210x1000 align:0}, op:{lowe}
7="Y 2000", status:{stack:0 proc:1}, sheet:{size:3210x1000 align:2000}, op:{full-cut}, prod:{size:2000x1000 piece:2}, remn:{size:1210x1000 scrap}
```

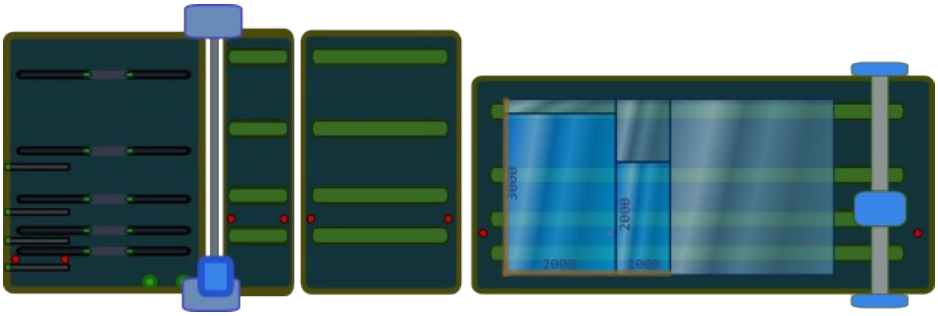


53 On *Strato* machines is common to have glass alignment errors in the order of the tenths of a millimeter which tend to deteriorate over time, as the blocks worn out.

9.5.3 Deletion of *Sheet* edges (L deletion)

As shown in the example above (see also in the example “Low-E material” above on p.96), when processing a low-E material on *Strato* the number of *Steps* increases due to “head deletions”, necessary to delete the low-E from the edges of the original *Sheet*.

A first approach is to perform the low-E deletion on *Float* just on the *Sheet* reference edges:



This is called “L deletion”: the *Strato* machine won’t have to perform any “head deletion” and will delete the low-E only on the cuts.

The advantages are:

- Very simple and predictable
- Easy coordination: exactly one *Job* for the *Float* and one *Job* for the *Strato*, very easy to resume
- No misalignment issues

The downsides are:

- The *Strato* machine will still have to delete the low-E on the cuts
- The deletions are performed with different tools and may appear a little different
- Doesn’t address structural deletion and *Shapes*

> ⚠ This is a mixed solution because some deletions are performed by the *Float* machine and others by the *Strato* machine, so the two low-E wheels should have similar characteristics.

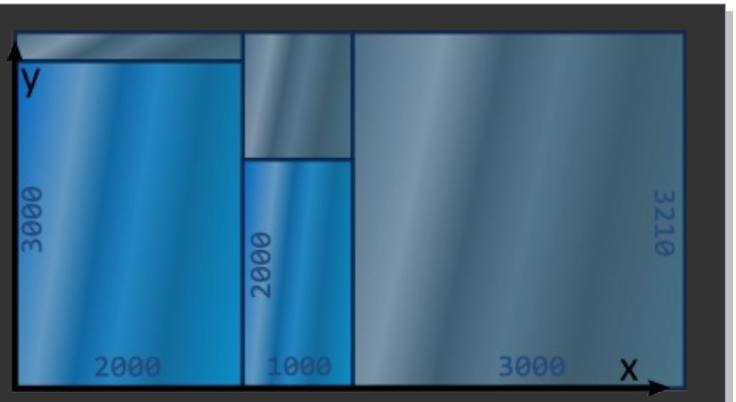
Scenario

The operator must process a low-E *Scheme* on a *Strato* machine. Since there is a combined *Float* machine equipped with a low-E wheel, the operator chooses to enable the “L” mode to avoid the *head deletions* on *Strato*.

The operator opens the *Project* and synchronize the *Scheme* on the *Strato* machine.

Adopting the simple example above, the *Strato HMI* sends to its machine this *Job*:

```
[project]
name="Lmode-strato"
scheme=1/1

[scheme]
type="strato"
size=6000x3210
glass-id="33.1LE"
glass-type=lowe
X=2000
  Y=3000 id:1
X=1000
  Y=2000 id:2

[pieces]
1={ size:2000x3000 }
2={ size:1000x2000 }

[steps]
1="X 2000", status:{stack:1 proc:0}, sheet:{size:6000x3210 extract align:2000}, op:{full-cut}, prod:{size:2000x3210 rotate:-90}, remn:{size:4000x3210 insert}
2="Y 3000", status:{stack:1 proc:1}, sheet:{size:3210x2000 align:3000}, op:{full-cut}, prod:{size:3000x2000 piece:1}, remn:{size:210x2000 scrap}
3="X 1000", status:{stack:1 proc:0}, sheet:{size:4000x3210 extract align:1000}, op:{full-cut}, prod:{size:1000x3210 rotate:-90}, remn:{size:3000x3210 scrap}
4="Y 2000", status:{stack:0 proc:1}, sheet:{size:3210x1000 align:2000}, op:{full-cut}, prod:{size:2000x1000 piece:2}, remn:{size:1210x1000 scrap}
```



As you can see is exactly equal to the normal work except the *Steps* of “head deletions”.

In the same time the *Float HMI* to sends to its machine the *Job* that performs low-E deletions on the *Sheet* reference edges:

```
[project]
name="Lmode-float"
scheme=1/1

[scheme]
type="float"
size=6000x3210
glass-id="33.1LE"
glass-type=lowe
X=2000
  Y=3000 id:1
X=1000
  Y=2000 id:2

[paths]
parameters = {
    size: 6000x3210
    trims: (0,0)
    tools: "lowe"
    lowe-width: 20
    lowe-margin: 0.5
   }
lowe=path:{
    start:{from:(0,0.5)}
    line:{to:(0,3209.5)}
   },
  path:{
    start:{from:(0.5,0)}
    line:{to:(3000,0)}
   }
```

Nothing stops the *HMI* to include labeling data (`[pieces]` and `[labels]` sections) to allow the *Pieces* labeling on the *Float* machine.

Note that the *HMI* was so clever to not extend the horizontal deletion on the *Sheet Scrap*.

All these operations are done when the operator synchronizes the *Scheme* on the *Strato* machine.

At this point both machines are have a *Job* synchronized: the operator will start first the processing on the *Float* machine, and when its done will then start the processing on the *Strato* machine.

To streamline the process, minimize downtime and avoiding human errors, consider the possibility of automate the start of *Strato* processing after the *Float* machine completes his *Job* as described below.

9.5.4 Incremental deletion by *Stripes*

The previous solution basically only saves the *Strato* machine to perform *Head deletions*. To achieve more on the *Float*, we have to address the misalignment problem describe above.

To mitigate this phenomenon, a solution is divide the entire *Scheme* into smaller parts, alternating the low-E deletions on the *Float* machine with the cuts on the *Strato* machine, thereby limiting the accumulation of errors.

Among the possible methods[54] to perform this division, each with its own benefits and drawbacks, the simple division by *Stripes* stands out due to its notable advantages:

- Fairly intuitive division, comprehensible by the machine operator
- Predictable coordination: each *Stripe* processed on *Float* is subsequently processed on *Strato*
- No risk to handle very small *Subsheets*

On the other hand, this solution it's not without flaws:

- This division mitigates misalignment only for the `X` cuts and not for the nested ones: more than four sibling `Y` or `Z` cuts may cause problems
- The work consists in a continuous back and forth between the two machines, so needs a supervisor to automate it
- Resuming a partial work is not easy, the *HMI* should provide a way to navigate between *Stripes*

Creating the Job on Strato

The `mac` files generated for the *Strato* machine for this strategy have the following characteristics:

- The main *Scrap* of each sub-*Scheme* is (except for the last *Stripe*) the next *Subsheet* processed on the *Float* machine, so in this case shouldn't be marked for disposal (`remn:{scrap}`) but instead inserted back (`remn:{insert}`)
- The low-E deletion on *Strato* machine must be totally excluded, there are two ways to do it:
  - Specify in each *Step*: `op:{full-cut,no-lowe}`
  - Specify once for the entire *Scheme*:

```
[scheme]
type="strato,no-lowe"
…
```

If the *Float* machine performs the low-E deletion also on the *Shapes*, it would be advisable to specify the low-E exclusion also in their definition (`[shape]` section), substituting `*=path{…}` with `no-lowe=path{…}` and removing possible pure low-E deletion paths like `lowe=path{…}`.

> This notation gives maximum flexibility to define any mixed low-E deletion combinations.

54 A rigorous method is described in the full specification: SP065 - Funzionamento in combinata.pdf

Scenario

Let's take the previous example and work it using this strategy.

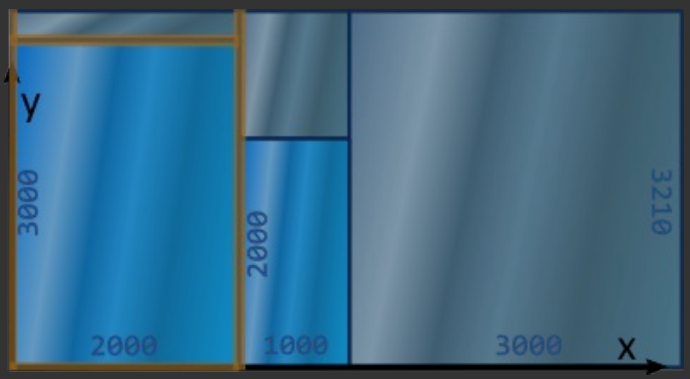
> ⚠ This combined work involves coordinating the two machines (see "Coordinating the workflow" below on p.147). While certain coordination dynamics will be hypothesized below for clarity, developers are under no obligation to implement them in a specific way; they have complete freedom to decide how to manage those mechanisms.

The operator must process a low-E *Scheme* on a *Strato* machine. Since there is a combined *Float* machine equipped with a low-E wheel, the operator chooses to enable the "deletion by *Stripes*" mode to avoid at all the low-E deletions on *Strato*.

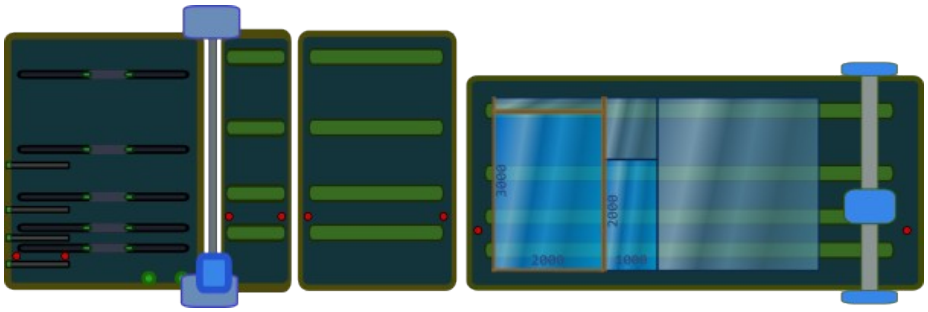
The operator opens the *Project* and synchronize the *Scheme* on the *Strato* machine.

The *Float HMI* synchronizes this *Job* (low-E deletion on *Stripe* 1):

```
[project]
name="stripe-deletion-float"
scheme=1/2

[scheme]
type="float"
size=6000x3210
glass-id="33.1LE"
glass-type=lowe
X=2000
  Y=3000 id:1

[pieces]
1={ size:2000x3000 }

[paths]
parameters = {
    size: 6000x3210
    trims: (0,0)
    tools: "lowe"
    lowe-width: 20
    lowe-margin: 0.5
   }
lowe=path:{
    start:{from:(0,0.5)}
    line:{to:(0,3209.5)}
   },
  path:{
    start:{from:(0.5,0)}
    line:{to:(2000,0)}
   },
path:{
    start:{from:(2000,0.5)}
    line:{to:(2000,3209.5)}
   },
  path:{
    start:{from:(2000,3000)}
    line:{to:(0.5,3000)}
   }
```

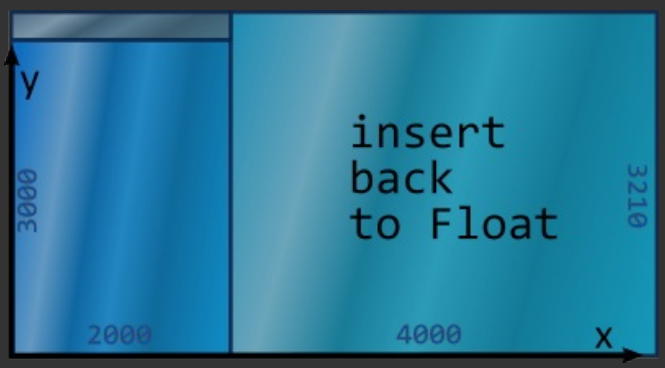


The operator starts the processing on the *Float* machine. Upon completion, the situation is as follows:

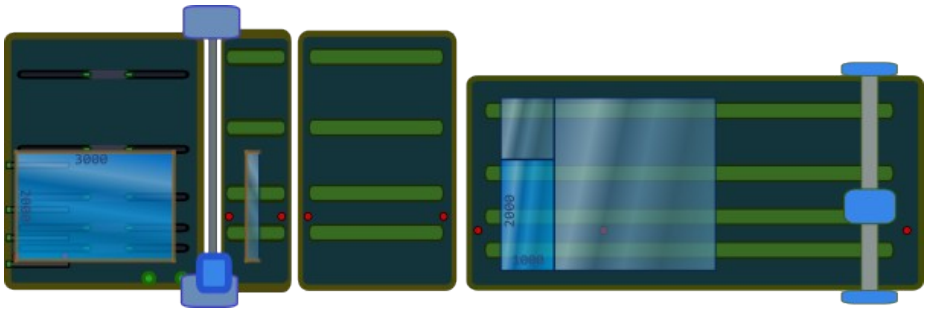


Now it's the turn of the *Strato* machine to proceed; its *HMI* sets this *Job* (cuts of *Stripe* 1):

```
[project]
name="stripe-strato"
scheme=1/2

[scheme]
type="strato,no-lowe"
size=6000x3210
glass-id="33.1LE"
X=2000
  Y=3000 id:1

[pieces]
1={ size:2000x3000 }

[steps]
1="X 2000", status:{stack:1 proc:0}, sheet:{size:6000x3210 align:2000}, op:{full-cut},
  prod:{size:2000x3210 rotate:-90}, remn:{size:4000x3210 insert} ; →back to Float
2="Y 3000", status:{stack:1 proc:1}, sheet:{size:3210x2000 align:3000}, op:{full-cut},
  prod:{size:3000x2000 piece:1}, remn:{size:210x2000 scrap}
```

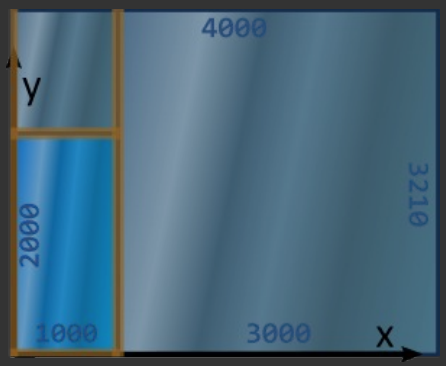


Some actor (operator or *Line manager*) gives the command to start the processing on the *Strato* machine. After the first *Step* the situation might be something like:



Now its time to proceed with the next (and last) Stripe; the *Float HMI* synchronizes this *Job* (low-E deletion on *Stripe* 2):

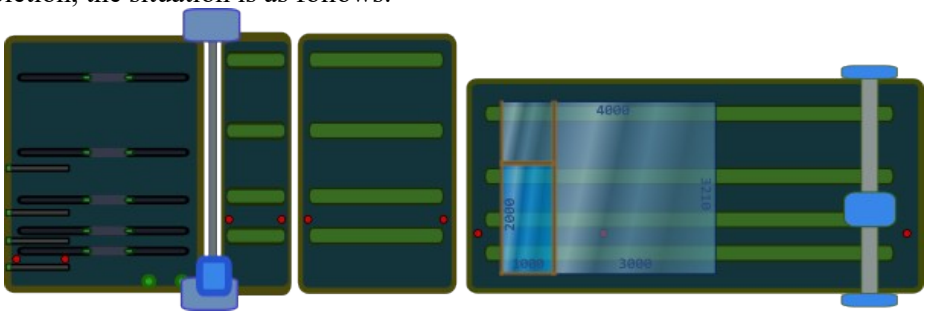
```
[project]
name="stripe-deletion-float"
scheme=2/2

[scheme]
type="float"
size=4000x3210
glass-id="33.1LE"
glass-type=lowe
X=1000
  Y=2000 id:2

[pieces]
2={ size:1000x2000 }

[paths]
parameters = {
    size: 4000x3210
    trims: (0,0)
    tools: "lowe"
    lowe-width: 20
    lowe-margin: 0.5
   }
lowe=path:{ start:{from:(0,0.5)} line:{to:(0,3209.5)} },
     path:{ start:{from:(0.5,0)} line:{to:(1000,0)} },
     path:{ start:{from:(1000,0.5)} line:{to:(1000,3209.5)} },
     path:{ start:{from:(1000,3000)} line:{to:(0.5,2000)} }
```



Some actor (operator or *Line manager*) gives the command to start the processing on the *Float* machine.

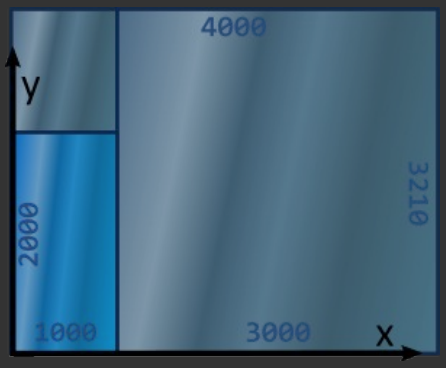
> Note that this may occur when the *Strato* machine is still working. The only constraint is that the *Remnant* of the first *Step* is already inserted back, this is guaranteed by having `step>1`.

Upon completion, the situation is as follows:



Once again, the *Strato HMI* synchronizes this *Job* (cuts of *Stripe* 2):

```
[project]
name="stripe-strato"
scheme=2/2

[scheme]
type="strato,no-lowe"
size=4000x3210
glass-id="33.1LE"
X=1000
  Y=2000 id:2

[pieces]
2={ size:1000x2000 }

[steps]
1="X 1000", status:{stack:1 proc:0}, sheet:{size:4000x3210 extract align:1000}, op:{full-cut},
  prod:{size:1000x3210 rotate:-90}, remn:{size:3000x3210 scrap}
2="Y 2000", status:{stack:0 proc:1}, sheet:{size:3210x1000 align:2000}, op:{full-cut},
  prod:{size:2000x1000 piece:2}, remn:{size:1210x1000 scrap}
```



After which the original *Scheme* is fully processed.

Coordinating the workflow

Regardless on how the original *Scheme* is divided, the mechanism for creating, transferring, and processing the sub-*Schemes* on the two machines remains unchanged.

The starting point is a *Scheme* for *Strato* machine involving low-E material, which must be deleted using the *Float* machine.

This *Scheme* will generate one or more pairs of sub-*Schemes*, the first for the *Float* machine (low-E deletion) and the second for the *Strato* machine (the cuts).

Upon selecting the original *Scheme* in the *HMI*, both sub-*Schemes* will be synchronized on the respective machine.

While the *Strato* process follows the *Float* chronologically, synchronizing both immediately allows the operator the flexibility to choose the starting point (ex. resuming a work interrupted the previous day).

Once the process of the sub-*Scheme* of the *Float* machine is complete, the processing of the corresponding sub-*Scheme* of the *Strato* machine can start.

The next *Sheet* that will be processed on *Float* machine is always the first *Remnant* inserted in *Feed*

by the *Strato* machine as final operation of the first *Step*. So after this operation, the *Float* machine may start to process its next sub-*Scheme* while the *Strato* machine is wrking the previous one. On the machines the next sub-*Scheme* should be synchronized immediately upon completion of the previous one.

Automatic starts

For safety reasons, the operator has to start the process manually, while subsequent starts may be automated: if the operator were required to manually start both machines each time, there would be a higher risk of downtime and accidentally starting the wrong machine at an inappropriate moment.

So to streamline the workflow it's beneficial to automatically start at least the processing on the *Float* machine. This ensures a workflow smooth and efficient, reducing idle time and improving overall productivity.

Since the machines do not have an overview of the entire workflow, the automation should be managed by a supervisor, who coordinates the workflow and ensures that the correct *Job* is uploaded/ synchronized into each machine, preventing costly errors.

A structure such as the one described in sect. 2.7.1 “Coordinating the combined workflow” above on p.17 may be the most natural way to organize the various actors.

The *HMI* can request the machine to start the processing as described in sect. 3.16.1 “Requesting a start” above on p.34.

Starting the Float machine

If simultaneous processing is feasible, the *Float* machine can start as soon as the *Strato* machine returns the first *Remnant* (`step>1` with `working==1`).

If the simultaneous processing is not allowed, or if the sub-*Scheme* on *Strato* has just one *Step*, the *Float* machine can start as soon as the *Strato* machine finishes its *Job* (`mach-status=="ready" && scheme-done==1`).

In either case, the start event is a single occurrence and must NEVER be repeated. In case of any failure, the operator can start the processing manually.

10 Monitoring machines

10.1 Objectives

Provide a description of the additional functions dedicated to machine monitoring.

10.2 Context

Collecting machine usage data is an increasingly need: measuring performance, identifying bottlenecks, and monitoring machine health to enable predictive maintenance are becoming ever more important for maximizing productivity.

10.3 Raw machine events

From version `2025-06` of MacoLayer an event detector component was added.

Machine events can be notified to clients (see 3.11 “Events notification” above at p.31), logged in a file, or signaled to an external sink via a socket with a certain payload (typically json).

Category	*Event*	*Arguments*	*Description*
Machine	`connected`	`machine:<name>`	Connected with the machine
	`disconnected`		Machine unavailable (off?)
	`status-changed`	`generic-status:<val>`	Machine generic status changed `UNKN` (unknown) `IDLE` (doing nothing) `ALRM` (blocked) `BUSY` (preparing) `WORK` (processing job) `WAIT` (waiting user)
	`alarms-changed`	`emg-list:<indexes>`	Blocking errors occurred
	`alarms-cleared`		No more blocking errors
	`message-set`	`<index>`	Machine showing a new notification
	`message-reset`	`<index>`	A notification was reset
HMI	`project-sent`	`prj-name:<name>` `schemes-count:<num>`	Project files sent to machine

	`job-selected`	`prj-name:<name>`
		`scheme:<index>`
		`step:<index>`
		`or`
		`manual:<dim>`
			A new job was sent to machine

	`job-cleared`		Job deselected on machine

	`material-selected`	`glass-id:<name>`
		`h-glass:<thickness>`
		`glass-type:<list>`
			Selected a material on machine

	`config-changed`	`work-selectors:<list>`
		`speed-override:<val>`
		`cut-recipe:<list>`
			A configuration that affects the processing time has changed

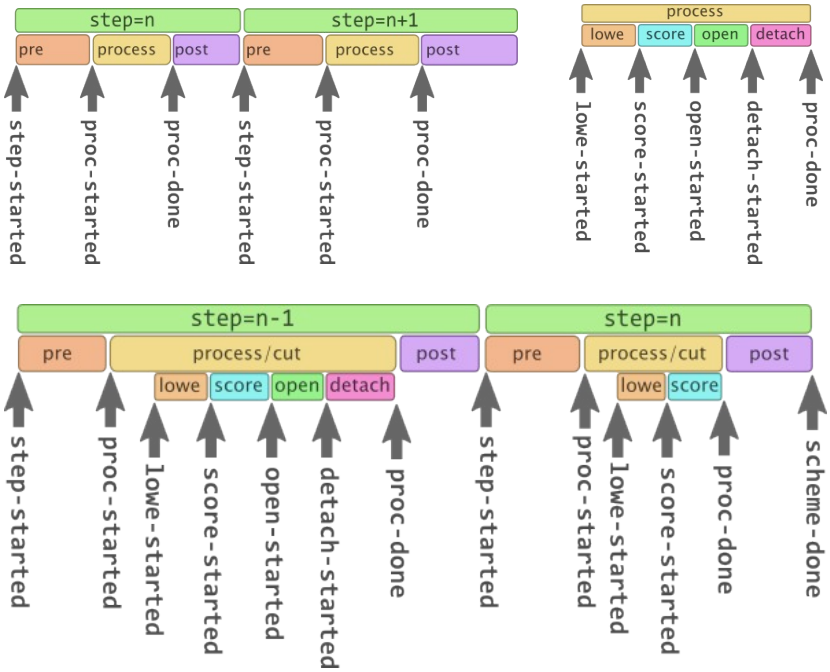
	`request-failed`		A request from HMI was denied (bad data?)

Category	Event	Arguments	Description
Job	scheme-started	prj-name:<name> scheme:<index> work-selectors:<list> speed-override:<val> cut-recipe:<list>	Starting to process a job (scheme/stripe)
	scheme-done	prj-name:<name> scheme:<index>	Completed a job (scheme/stripe)
	step-started	prj-name:<name> scheme:<index> step:<index> step-data:{...}	Starting the execution of a step (Strato)
	proc-started		Starting the glass processing
	proc-done		Glass/cut successfully processed
	proc-failed		Glass/cut processing error
	tpf-started		Starting the TPF removal (Float)
	lowe-started		Starting the low-E removal stage
	score-started		Starting the scoring stage
	open-started		Starting the opening stage (Strato)
	detach-started		Starting the detach stage (Strato)
	deliver-started		Delivering a piece (Strato)
	deliver-done		Delivered a piece (Strato)
	piece-taken	piece-taken:<id>	User retrieved delivered piece
	sheet-loaded		Loaded a new sheet on table
	sheet-out		A sheet was moved out from table
External	user-button	user-buttons:<list>	User pressed or released a button
	user-login	<username>	A user logged in
	user-logout	<username>	A user logged out
	piece-broken	prj-name:<name> scheme:<index> step:<index> piece:<index>	User declaring a broken piece
	maintenance	<action>	Maintenance action

Events may be logged in a csv file

```
2025-06-04T16:58:30+02:00,connected,"machine=Strato-Dummy"
2025-06-04T16:58:30+02:00,status-changed,"generic-status=IDLE"
2025-06-04T16:58:42+02:00,project-sent,"prj-name=~strato-test-prj","schemes-count=3"
2025-06-04T16:58:42+02:00,material-selected,"glass-id=33.1","h-glass=3/0.38/3","glass-type=strato"
2025-06-04T16:58:42+02:00,job-selected,"prj-name=~strato-test-prj","scheme=1","step=1"
2025-06-04T16:58:44+02:00,status-changed,"generic-status=WORK"
2025-06-04T16:58:44+02:00,scheme-started,"prj-name=~strato-test-prj","scheme=1"
2025-06-04T16:58:44+02:00,step-started,"prj-name=~strato-test-prj","scheme=1","step=1"
2025-06-04T16:58:46+02:00,proc-started
2025-06-04T16:58:46+02:00,score-started
2025-06-04T16:58:47+02:00,open-started
2025-06-04T16:58:47+02:00,status-changed,"generic-status=ALRM"
2025-06-04T16:58:47+02:00,alarms-changed,"emg-list=123,30800012"
2025-06-04T16:58:47+02:00,proc-failed
2025-06-04T16:58:49+02:00,status-changed,"generic-status=IDLE"
2025-06-04T16:58:49+02:00,alarms-cleared
2025-06-04T16:58:50+02:00,status-changed,"generic-status=WORK"
2025-06-04T16:58:50+02:00,step-started,"prj-name=~strato-test-prj","scheme=1","step=1"
2025-06-04T16:58:51+02:00,proc-started
2025-06-04T16:58:53+02:00,detach-started
2025-06-04T16:58:54+02:00,proc-done
2025-06-04T16:58:58+02:00,status-changed,"generic-status=WAIT"
2025-06-04T16:59:03+02:00,status-changed,"generic-status=WORK"
2025-06-04T16:59:03+02:00,piece-taken,"piece-taken=1"
2025-06-04T16:59:04+02:00,step-started,"prj-name=~strato-test-prj","scheme=1","step=3"
2025-06-04T16:59:05+02:00,status-changed,"generic-status=BUSY"
2025-06-04T16:59:06+02:00,status-changed,"generic-status=IDLE"
```

In case of *Strato* machine the steps sequence and the processing phase are characterized by the events:



11 Appendices

11.1 Notes about *HMI* development

11.1.1 Responsibilities

The *HMI* has the following functions:

- Display the machine's status to the operator (messages, current operation, etc.)
- Monitoring the machine status
  - Active alarms/messages
  - Selected job
    - Current material, current work selectors/settings
    - Last uploaded/ synchronized *Project*, current selected *Scheme*/*Step*
    - Execution status (current operation, completion percentage, remaining time, etc...)
  - Other statuses (glass position, indication lights, axes positions, etc.)
- Prepare and load a job onto the machine
  - Entering a manual cut
    - Selecting a material for the manual cut
    - Entering cut dimensions
    - Manual selection of desired cutting sequence
  - Automatic processing of one or more *Schemes*
    - Project transmission
    - Selecting the *Scheme*/*Step*
- Modify work parameters (speed override, work selectors, etc.)
- Machine commands (reset emergencies, etc.)
- Serving prints for *Piece* labels
- Read and store the specific characteristics of the machine (on connection or occasionally)
  - Table dimensions, modules presence
  - Limits (cut range, movements, rotations, etc.)
  - Other characteristics (available tools, preferred rotations, etc.)

11.1.2 Getting started

Here's how to start to develop a program that interacts with our machines:

- Grab the latest MacoLayer in your test environment
  - `C:\> mkdir Macotec`
  - `C:\> cd Macotec`
  - `C:\Macotec> git clone https://github.com/matgat/MacoService.git`
- Run `MacoLayer.exe`
- Check the connection port set in `MacoLayer.ini`
- Ask MACOTEC what to do in case of authentication problems
- Interact with the emulated machines

11.1.3 General principles

The first important concept is this: the *HMI* must reflect the actual state of the machine, otherwise it becomes a safety problem. So it is absolutely necessary to avoid showing invalid or untruthful information.

The *HMI* must therefore have the following states: the disconnected status (the program must obviously start in this state) and the connected status, during which the machine is monitored (messages, ...) and the relative information is displayed; the connected state is in turn divided into two sub-states: not synchronized (manual cuts, *Project* navigation) and synchronized (monitoring of the current transferred work).

The *HMI* must clearly show the operator the current state:

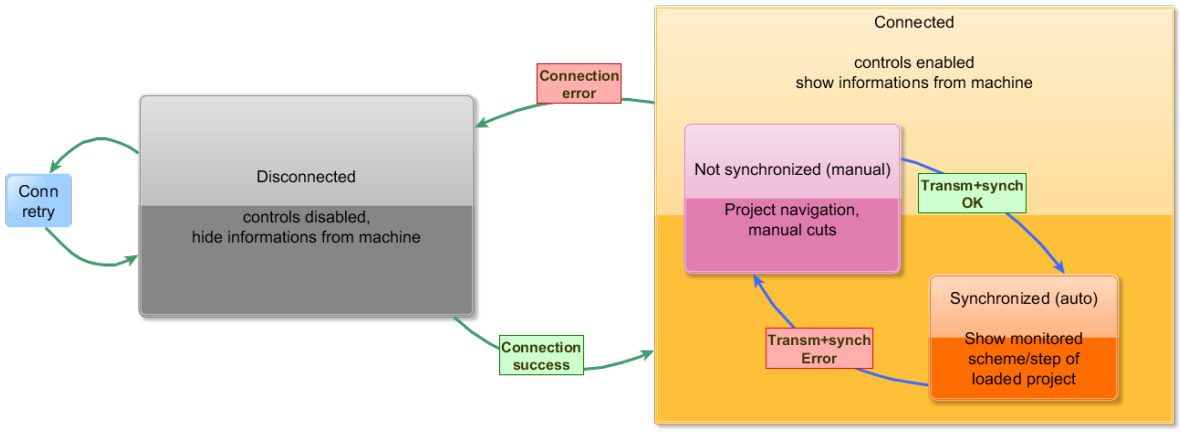


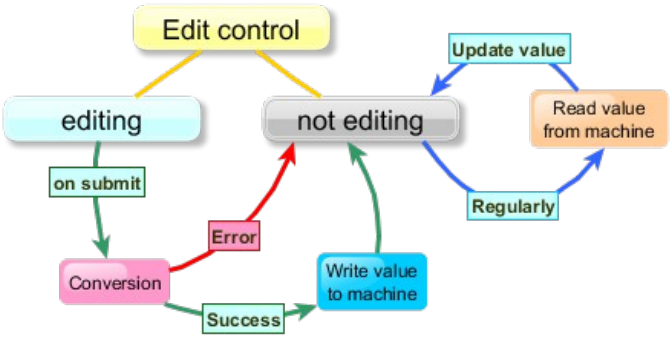
Figure 8: HMI states

> ⚠ If there's no connection any information relating to the machine is false and therefore should not be displayed; for the same principle, if the monitored state is not "fresh" (ex. older than 10s) may no longer be true and therefore should be hidden.

Controls that write a resource (ex. override speed) are disabled in the disconnected state.

In the connected state the controls reflect the current state of the resource they refer (show the current actual value).

When they are in the editing state they allow the value modification; when the new value is confirmed, if correct it is written on the machine, otherwise ignored (the value displayed will be constantly updated by the reading of the actual value on the machine).



11.2 Extended `json-ini` syntax

The text files format adopted for our descriptors is based on the `ini` format, so `key=value` pairs grouped into sections:

```
[section]
key=value
```

This base structure is enhanced by allowing values to be `json` blocks:

```
key=value,jsonkey:{json-block}
key=jsonkey:{json-block}
key={json-block}
```

> The adopted `json` *parser* is not *standard*: supports unquoted key/values, optional comma separators, tuples, line comments, …

11.2.1 Inconsistencies and pitfalls

While this combination allows taking advantage of the terseness of `ini` and the expressiveness of `json`, it also requires extra care.

The main inconsistencies arise from:

- The comma acts as an optional separator in `json`, while it has no particular meaning in `ini`
- A line break indicates a new element in `ini`, whereas it's just a space in `json`
- Different syntax for comments

Values containing commas

Consider:

```
tools1="lowe,score"
```

The value of `tools1` is the string: `lowe,score`.

Now consider:

```
tools2=lowe,score
```

The value of `tools2` is a `json` block containing two empty keys `lowe` and `score`: this was likely not intended.

Concatenation of `json` blocks

A trailing comma allows to expand the definition of a not nested `json` block on multiple lines:

```
inikey=jskey1:{ a:1 b:2 },
       jskey2:{ c:3 } ; ok
```

Unfortunately, the significance of line breaks in `ini` files makes the following form problematic:

```
inikey=jskey1:{ a:1 b:2 }
      ,jskey2:{ c:3 } ; bad
```

Improper handling of these commas like in the previous example leads to incomprehensible parsing errors.

Note that commas can always be omitted within `json` blocks nested in curly braces, also when the block spans on multiple lines:

```
inikey={jskey1:{ a:1 b:2 }
       jskey2:{ c:3 }} ; ok
```

Comments

```
inikey1=jskey:{ a:1 b:2 } ; ini comment
inikey2=jskey:{ a:1 b:2 } // json comment
```

For consistency better always adopt the `ini` comment (`;`), also supported within `json` blocks.

11.3 Mapping table for old *Strato* material parameters

In the transition from old Fagor *Strato* machines, it is possible to reuse the old dialog for editing the *Strato* material parameters using the following map:

Fagor	Original caption	Field	New caption [ita]
R717	Tipologia di vetro monolitico	sheet-type	*[lowe,float,lami,strato,filmsup,filminf]*
		h-glass	*Spessori vetro*
R722	Timeout Cicli di Separazione	wheel-angle	*Gradazione rotelle di incisione*
		cut-recipe	*Ricetta di taglio*
R704	Pressione rotella tg inferiore	f-score-inf	*Spinta testina di incisione inferiore*
R703	Pressione rotella tg superiore	f-score-sup	*Spinta testina di incisione superiore*
R716	Velocità di taglio	v-score	*Velocità di incisione*
		f-lowe	*Spinta mola basso emissivo*
R723	Velocità molatura	v-lowe	*Velocità mola basso emissivo*
		h-brkbar	*Altezza barra di troncaggio*
		t-brkbar	*Tempo barra di troncaggio*
R507	Contro pressione premilastra	f-clamps-brk	*Spinta premil durante apert con barra*
R707	Press rotella apertura std	f-brkwhl-sup	*Spinta rotella di apertura singola*
R724	Velocità rotella apertura std	v-brkwhl-sup	*Velocità rotella di apertura singola*
R709	Press rotella superiore rifilo	f-brkwhl-sup-trim	*Spinta rotella sup apertura rifilo*
R711	Press contro-rotella superiore rifilo	-	
R710	Press rotella inferiore rifilo	f-brkwhl-inf-trim	*Spinta rotella inf apertura rifilo*
R712	Press contro-rotella inferiore rifilo	-	
R713	Velocità apertura rifilo	v-brkwhl-trim	*Velocità rotelle di apertura rifilo*
R506	Pressione premilastra	f-clamps-max	*Massima spinta premilastra ammessa*
R705	Primo movimento stacco	f-dtch-pull	*Trazione del plastico durante la sep*
R706	Secondo movimento stacco	v-dtch-pull	*Velocità traz plastico durante la sep*
R721	Timer preriscaldamento lametta std	t-heat-pre	*Tempo riscald prima di tirare*
R719	Timer riscaldamento PVB nel ciclo lametta	t-heat-pull	*Tempo riscald durante trazione*
R708	Timer riscaldamento ciclo std	t-heat-add	*Tempo riscald addiz per ciclo rifilo*
R718	Tempo riscaldamento taglio	t-heat-cut	*Tempo riscald taglio cicliche speciali*
R714	Tempo lamp stacco rifilo con strappo	-	