

H_Ard_NG50_Base

MIDI music generator for Arduino Mega / Due

- Startup and control
- Time, ticks and BPM
- Structure of a song
- Patterns, players and effects
- Automatic composer

This manual describes the behaviour of the software as verified in a PC simulation. Anything that depends on the real hardware (display, network, keys, real-time timing) must be checked on the board.

Contents

Introduction.....	4
Purpose of this manual.....	4
Architecture at a glance.....	4
Conventions used in this manual.....	4
Verification status.....	4
1 Startup, start/stop and keys.....	5
1.1 Initial state.....	5
1.2 Start and stop.....	5
1.3 Key modes.....	5
1.4 Power-up protection.....	6
1.5 Commands from the core unit serial port.....	6
2 Optional features (config.h).....	7
3 Play lists and memory.....	8
3.1 playList and endList.....	8
3.2 Sizes.....	8
3.3 What happens when a list is full.....	8
3.4 Sizing the lists.....	8
3.5 Notes on the PlayEndLib library.....	8
4 Diagnostics on the serial port (debug).....	9
4.1 Enabling.....	9
4.2 Messages.....	9
4.3 Reading “Tick max us”.....	9
5 Time: ticks, BPM, durations.....	10
5.1 The tick.....	10
5.2 BPM and bars.....	10
5.3 Positions and durations.....	10
5.4 Accuracy.....	10
6 Anatomy of a song.....	11
6.1 Basic structure.....	11
6.2 Main variables.....	11
6.3 Splitting the song into sections.....	11
7 Song parameters.....	12
7.1 Key and scale.....	12
7.2 Volume.....	13
7.3 Channels.....	13
7.4 Random control changes.....	13
7.5 Ending a song.....	13
8 Rests (empty notes).....	14

8.1 Rests in patterns.....	14
8.2 Example.....	14
8.3 How rests behave.....	14
8.4 Generated rests.....	14
8.5 Technical notes.....	15
9 Chords.....	16
9.1 Chords from a pattern.....	16
9.2 Chords from an array.....	16
10 Patterns: writing and generating them.....	18
10.1 The basePat array.....	18
10.2 Reading and writing.....	18
10.3 Generators.....	18
10.4 Variations on an existing pattern.....	19
11 Players and effects.....	20
11.1 Common parameters.....	20
11.2 Players.....	20
11.3 Effects.....	20
11.4 Free modules.....	21
12 The automatic composer (complexp).....	22
12.1 The complexp structure.....	22
12.2 Effect on the song.....	22
12.3 Sections and tracks.....	22
13 Complete song example.....	24
13.1 The code.....	24
13.2 How it works.....	24
13.3 Result.....	24
13.4 Quick recipes.....	25
14 Choosing the active song.....	26
15 Things to know.....	27
Appendix A Troubleshooting.....	28
Appendix B Glossary.....	29

Introduction

Purpose of this manual

This manual explains how to use and program `H_Ard_NG50_Base`, an Arduino sketch that generates music and sends it as MIDI messages. It covers startup and commands, how time is kept, the structure of a song, the tools for writing or generating patterns, and the players that turn patterns into notes.

It is intended for anyone who wants to write their own song or modify the existing ones. Basic knowledge of the Arduino IDE and of the C/C++ language is assumed.

Architecture at a glance

Everything revolves around a single concept: the **tick**, the smallest unit of time. On every tick the system runs the same chain of operations:

Step	What happens
1. Tick	The scheduler (<code>micros()</code> , <code>Scheduler.h</code>) fires the tick at the interval computed from the BPM.
2. <code>carica()</code>	Increments <code>contick</code> and calls the function of the active song (chapter 14).
3. Song	Decides what to play based on <code>contick</code> : at the start it sets up tempo, key and patterns, then calls the players on every bar.
4. Players	Read the patterns in <code>basePat</code> and queue notes with <code>playAdd(nota, midichan, volume, durata, inizio)</code> .
5. <code>playList</code>	Holds the waiting notes: when the start tick arrives, the note is played.
6. <code>endList</code>	Holds the sounding notes: at tick <code>inizio + durata</code> the note is switched off.

All positions and durations are expressed in ticks. Patterns do not contain absolute notes but **scale degrees**: this is why the same pattern is transposed automatically when the key changes.

Conventions used in this manual

Notation	Meaning
<code>functionName()</code>	names of functions, variables, files and macros in the code
Dark block	code example to copy into the sketch
NOTE	important note or non-obvious behaviour
WARNING	situation that may cause malfunctions or unexpected results
TIP	practical advice
(see 7.5)	reference to a chapter or section of the manual

Identifiers in the code (functions, variables, parameters) are in Italian and are quoted exactly as they appear in the sketch.

Verification status

Everything described here has been checked against the code by simulating it on a PC with simulated hardware (ticks, notes, rests, chords, keys at startup, automatic start). Display, network, sensors and real-time timing have not been tested: they must be checked on the board.

1 Startup, start/stop and keys

This chapter describes how the system powers up, how a song is started and stopped, and which commands are available from the keys or from the serial port.

1.1 Initial state

The variable `boolean acceso` (in `H_Ard_NG50_Base.ino`) determines what happens at power-up:

Value	Behaviour
<code>true</code>	the song starts by itself at the end of <code>setup()</code>
<code>false</code>	the system stays idle until start is pressed

In either case `setup()` first puts the system in the “off” state by calling `fermaTutto()` (as the old `OnOff()` used to do in setup). Only at the end, if `acceso` was `true`, does it call `OnOff()`, i.e. start. This way the system always starts from a clean state, with no hanging notes.

1.2 Start and stop

`OnOff()` toggles the state: if the song is stopped it starts it, if it is running it stops it.

Action	What it does
Start	if <code>contick == 0</code> it calls <code>carica()</code> to prepare the song, sends <code>MIDISTART</code> and enables the tick.
Stop	<code>fermaTutto()</code> : stops the ticks and switches off all notes.

1.3 Key modes

The mode is selected with `KEYS_MODE` in `config.h` (see chapter 2).

KEYS_FULL – full keypad

Keys are read by `ioi.readKey()`, keypad type 3 “4 KEY”.

Key	Label	Function
4	E	deferred end: <code>diffstop = true</code> (see 7.5)
5	F	start/stop
6	select	start/stop
8	N	not used
9	C	enable/disable random control changes (see 7.4)
10	A/M	not used
11	X	reset values: the first press arms it, a second press within 3 seconds resets everything

KEYS_ONOFF – single key

A single key on pin `PIN_ONOFF` (`config.h`, default 7), wired to ground with the internal pull-up: it works as start/stop.

WARNING Set `PIN_ONOFF` to match the actual wiring of the board.

KEYS_NONE – no keys

No keys connected: the song starts only with `accesso = true` or with a command from the core unit serial port.

1.4 Power-up protection

During the first **500 ms** after startup the keys are ignored: the value read is only used as the initial state. This way a “stuck” key or electrical noise at power-up cannot start the song by mistake. In debug mode every key press prints **Tasto n** on the serial port.

1.5 Commands from the core unit serial port

With `USE_NETWORK` enabled, the core unit (`Serial1`) can send text commands:

Command	Effect
start	starts the song
stop	stops the song
dstop	deferred end (see 7.5)

2 Optional features (config.h)

Some parts of the system can be excluded from compilation. This saves memory and, above all, processing time: on slow boards the display or the network can steal time from the tick.

In `config.h` each option is `1` (enabled) or `0` (excluded). An excluded option uses neither memory nor time: the calls disappear together with their arguments, so not even the `String` objects passed as parameters are built.

Option	Description
<code>USE_DISPLAY</code>	LCD 5110 display (IOInt library). The calls are the macros <code>LCD_print</code> , <code>LCD_printNumI</code> , ... The display is refreshed only if something has changed and if there is time left before the next tick.
<code>USE_NETWORK</code>	Socket / OSC / I2C to the external display and modules (ExtMem), serial link with the core unit, external memory. Even when compiled in, messages are sent only if <code>FunStatus[4]</code> is not ' ' (Mega: 'S' = socket).
<code>USE_SENSORI</code>	Light, sound and motion sensors (ExtSensori). In the current code they are only initialised.
<code>KEYS_MODE</code>	<code>KEYS_NONE</code> / <code>KEYS_ONOFF</code> / <code>KEYS_FULL</code> (see 1.3).

The options are guarded by `#ifndef`: besides editing `config.h`, they can be forced from outside with `-D` at compile time.

TIP If the music cannot keep up, first try setting `USE_DISPLAY 0` and `USE_NETWORK 0` (see also chapter 4).

3 Play lists and memory

Notes are not played at the moment a player generates them: they are queued and played when their tick arrives. To do this the system uses two lists.

3.1 `playList` and `endList`

List	Contents	Size
<code>playList</code>	notes to be played, waiting for their start tick	8 bytes per note
<code>endList</code>	sounding notes, to be switched off at the end of their duration	4 bytes per note

They are static arrays of the `PlayEndLib` library: the space is always reserved, even when the lists are empty.

3.2 Sizes

The sizes are set in `config.h` with `PLAYELE` (`playList`) and `ENDELE` (`endList`):

Board	PLAYELE	ENDELE
Mega	96	32
Due	256	128

3.3 What happens when a list is full

- If `playList` is full, the extra note is discarded and `'0'` (overflow) is shown.
- If `endList` is full, the note just played is switched off immediately, so that no notes are left hanging.

3.4 Sizing the lists

In debug mode, on stop, `Max play/end: a/b` is printed, i.e. the peaks reached by the two lists. Values measured in simulation:

Case	<code>playList</code>	<code>endList</code>
Test songs	24	3
Full composer (all tracks, base 32)	64	19

TIP Set the sizes slightly above the measured peak: the RAM saved remains available for patterns (see chapter 15).

3.5 Notes on the `PlayEndLib` library

- The library is separate (`librariesDev/PlayEndLib`). Its template parameter is called `N`, not `PLAYELE`, because `PLAYELE` and `ENDELE` are project macros.
- Function signatures use `PlayListT` (typedef in `config.h`), because the Arduino IDE does not generate prototypes when a number appears between `<` `>`.
- `deleteCurrent()` does not preserve order (the last element takes the place of the deleted one); to preserve it there is `deleteCurrentOrdered()`. The project does not need it.

4 Diagnostics on the serial port (debug)

Debug mode is used to see on the serial port what the system generates and to measure whether processing time is sufficient.

4.1 Enabling

Set `boolean debug = true;` in `H_Ard_NG50_Base.ino` and open the serial monitor at **9600** baud.

WARNING In debug mode MIDI messages are sent as text ("`Cmd: 144 - Nota 60 - Vel. 100`") and NOT as MIDI bytes: the connected instrument does not play. This mode is only for checking what is generated.

4.2 Messages

The messages are printed in Italian, exactly as shown below.

Message	Meaning
Status: 1 / Status: 0	start / stop
Max play/end: a/b	list peaks since the last start (printed on stop)
Tick max us: n	duration of the longest tick, in microseconds (printed on stop)
Tasto n	key pressed (KEYS_FULL only)
Spengo TUTTE le note	"switching off ALL notes": stop
Invio dati generali	"sending general data": status update to the outside (only with <code>accesso = true</code>)

4.3 Reading "Tick max us"

The time available for one tick is:

```
tick_interval_us = 60,000,000 / (BPM x 16)
```

Tick max us tells how long the heaviest tick took. If it gets close to the interval or exceeds it, the system cannot keep up and the notes arrive late. Remedies, from the simplest:

- exclude display and network (chapter 2);
- make the song lighter (fewer tracks or effects per bar);
- lower the BPM.

Example: at 120 BPM a tick lasts 31,250 μ s; a **Tick max us** of 28,000 means the margin is almost used up.

5 Time: ticks, BPM, durations

5.1 The tick

The song is paced by the tick. A tick lasts $60 / (\text{BPM} \times 16)$ seconds (`beatdiv = 16`): a reference bar is divided into 16 ticks.

BPM	Tick duration	16-tick bar
20	187.5 ms	3 s
30	125 ms	2 s
60	62.5 ms	1 s
120	31.25 ms	0.5 s

5.2 BPM and bars

- A 16-tick bar lasts $60 / \text{BPM}$ seconds: with `base1 = 16` the BPM therefore equals bars per minute.
- With `base1 = 32` the bar lasts twice as long.
- The BPM is `PotAverage[9]` (10..240). It can be changed while playing (`PotAverage[9] = 40;`): `loop()` updates the interval and the change takes effect immediately.

5.3 Positions and durations

Every position and every duration is in ticks. A note with start `i` and duration `d` starts at tick `i` and stops at tick `i + d`.

Example: a duration of 2 ticks at 30 BPM corresponds to $2 \times 125 \text{ ms} = 0.25 \text{ s}$.

5.4 Accuracy

The tick is timed with `micros()` (`Scheduler.h`) and does not accumulate drift: if one cycle is late, the next one catches up. If more than one tick in a row is lost, the scheduler restarts from the current time instead of running a burst of overdue ticks.

6 Anatomy of a song

A song is an ordinary C++ function called on every tick by `carica()`, right after `contick++`. The function does not “play” directly: on each call it decides whether it is time to prepare something and queues the notes.

6.1 Basic structure

```
if (contick == 0) { ... }           // initialisation, once only: tempo, key, patterns
if (contick % base1 == 0) { ... }  // at the start of every bar: play the patterns
```

The first block runs only once, on the first tick; the second at the start of every bar. Between bars the function is still called but does nothing: the notes already queued are played from the lists.

6.2 Main variables

Variable	Meaning
<code>contick</code>	tick counter since the start. It is unsigned: on AVR (Mega) it is 16-bit and wraps to 0 after 65535; on the Due it is 32-bit.
<code>base1</code>	ticks per bar (16 or 32 in the examples).
<code>tickDivider(n, base1)</code>	bar number, from 0 to n-1, within a cycle of n bars.
<code>DatiVideo.*</code>	<code>baselen</code> , <code>songlen</code> , <code>bitfull</code> , <code>flagmodifica</code> : used only by the display and external units (ticks per bar, bars in the song, current bar, “needs refresh”).

NOTE Use powers of 2 for `base1` and for the number of bars: that way, when `contick` wraps to 0, the song cycle is not broken in the middle.

6.3 Splitting the song into sections

`tickDivider()` lets you organise the song into sections that repeat cyclically:

```
bitarcf = tickDivider(64, base1); // 0..63, then wraps
if (bitarcf <= 3) { /* intro */ } else if (bitarcf <= 19) { /* verse */ } ...
```

In this example the song lasts 64 bars: the first 4 are the intro, bars 4 to 19 the verse, and so on.

7 Song parameters

The main parameters of a song are held in the **PotAverage** array (the “pot” variables, which in the versions with potentiometers correspond to the knobs).

Index	Meaning
PotAverage[1]	control change channel (7.4)
PotAverage[2]	rhythm row (0..7), for the rhythm engine
PotAverage[3]	key and scale (7.1)
PotAverage[7]	next key (display only)
PotAverage[8]	base volume, 0..127 (default 100) (7.2)
PotAverage[9]	BPM, 10..240 (chapter 5)
[0], [4], [5]	not used

7.1 Key and scale

Key and scale are encoded in a single number and activated with **setModeKeySong()**:

```
PotAverage[3] = row * 12 + note;      setModeKeySong(PotValue(3));
```

note is the tonic:

0	1	2	3	4	5	6	7	8	9	10	11
C	C#	D	D#	E	F	F#	G	G#	A	A#	B

row is the scale type (**scaleMod** table in **config.h**):

Row	Scale	Row	Scale
0	major (Ionian)	10	melodic minor
1	minor	11	harmonic minor
2	Dorian	12	harmonic major
3	Phrygian	13	oriental
4	Lydian	14	Hungarian major
5	Mixolydian	15	enigmatic
6	Aeolian	16	major pentatonic
7	Locrian	17	diminished
8	Neapolitan	18	whole tone
9	Neapolitan minor	19	chromatic (all semitones)

Value	Calculation	Result
0	$0 \times 12 + 0$	C major
12	$1 \times 12 + 0$	C minor
14	$1 \times 12 + 2$	D minor
24	$2 \times 12 + 0$	C Dorian

After **setModeKeySong()**, **keyArray[]** (the notes of the scale) and **note4key** (how many there are) are ready.

NOTE Patterns store scale degrees, not notes: when the key changes they are transposed automatically.

7.2 Volume

`PotAverage[8]` is the base volume. To make the performance less mechanical, the players vary it randomly by 20% once every 3 notes (`volumePlus(volume, 20, 3)`).

The “full” versions of some players also accept `startvolume` and `diffvolume`: the first is the initial volume, the second the increment per note, positive for a crescendo and negative for a diminuendo.

```
moduloBase(0, 16, 0, 0, 1, 40, 4);    // starts at 40 and rises by 4 on every note
```

7.3 Channels

`midichan`, the last parameter of almost every player, is the MIDI channel 0..15 (0 = channel 1) and also the track tag. A note is merged with one already sounding only if they have the same tag and the same pitch.

TIP Use different channels for different tracks: besides being able to assign them to different instruments, this prevents identical notes from different tracks from merging.

7.4 Random control changes

They are enabled with key 9 (`FunStatus[1] = 'C'`). They send 8 controllers (0..7) with random values every `60/BPM` seconds, on channel `PotAverage[1]`.

7.5 Ending a song

To end a song at the right moment instead of cutting it off abruptly, use the deferred end: `diffstop = true` (or key E, or the `dstop` command from the core unit). `carica()` stops the song on the next tick.

NOTE `carica()` contains two deferred-stop lines; the first one always fires first, so the second one (end on multiples of 1024 ticks) is never reached. If you want the song to end on multiples of 1024 ticks, comment out the first one.

8 Rests (empty notes)

Silence is as much a part of music as notes. In `H_Ard_NG50_Base` a rest is obtained simply by not placing any note in a given interval; inside patterns it is marked with a special value.

8.1 Rests in patterns

In `basePat` a rest is an element with interval **255** and octave **99**. It can be written in two equivalent ways:

```
nota2int(255, i);           // with the special MIDI value 255
int2int(255, 99, i);        // with degree and octave
```

The players (`moduloBase` and the others) skip these elements: no note is queued, so there is silence at that time.

8.2 Example

```
nota2int(60, 0); nota2int(60, 1);    // C C
nota2int(255, 2); nota2int(255, 3);  // 2-tick rest
nota2int(62, 4);                     // D
int2int(255, 99, 5);                 // 1-tick rest
nota2int(64, 6); nota2int(64, 7);    // E E
moduloBase(0, 8, 1);                 // play the pattern on channel 1
```

Result: C from tick 0 to 2, D from 4 to 5, E from 6 to 8. Two consecutive slots with the same note become a single note 2 ticks long.

8.3 How rests behave

- A rest between two identical notes keeps them separate (`C _ C` = two notes); without a rest they merge (`C C C` = one note 3 ticks long).
- A rest at the start delays the attack; a rest at the end leaves silence until the end of the pattern.
- If nothing is programmed in an interval there is already silence: a note shorter than its slot leaves the rest of the slot silent.
- For chords and arpeggios: `fastloadPlay(255, durata, inizio, ...)` is a rest.

8.4 Generated rests

To insert a random rest into an already full pattern:

```
fxRandomPauseSingle(starti, patlen, ltick, startpos)
fxRandomPauseSingle(0, 8, 2, 3);           // 2 ticks of silence starting at tick 3
```

Parameter	Special value
<code>ltick</code>	0 = random length (2 or 4)
<code>startpos</code>	-1 = random position

Some generators (`baseFullControl`, `baseFibonacci`) have the `modoblank` parameter:

modobLank	Effect
0	random
1	no rests
2	half the note silent
3	the whole note silent

8.5 Technical notes

- Rests take no space in `playList`: they neither slow down nor fill the list.
- The functions that check the “notes still playing” on a channel (`contaPlayTick`, used by `moduloArpGlass` and `moduloMedia`) do not count rests: they may therefore restart slightly earlier, when only silence is left.
- `fxRandomPauseMulti` is marked “NON FUNZIONA” (does not work) in the code comments and has not been modified.

9 Chords

Chords can be obtained in two ways: by turning every note of a pattern into a chord, or by building a chord in an array and playing it with one of the available modes.

9.1 Chords from a pattern

Some players accept **modoaccordo** and **tipoaccordo**: each note of the pattern becomes a chord built on that note.

modoaccordo	Meaning
0	single note (no chord)
1, 2, 3 ...	1 = major scale, 2 = minor, 3 = Dorian ... (row of scaleMod = modoaccordo - 1, see 7.1)

tipoaccordo	Chord	tipoaccordo	Chord
0	random	7	augmented ninth
1	Maj/Min (triad)	8	minor ninth
2	major seventh	9	eleventh
3	seventh	10	augmented eleventh
4	added ninth	11	thirteenth
5	major with ninth	12	sixth
6	ninth		

Example: every note becomes a major triad on channel 3.

```
moduloBase(0, 16, 1, 1, 3); // C -> 60,64,67 D -> 62,66,69
```

NOTE The triad is built on the note (D major = D F# A) and is not necessarily within the key of the song.

9.2 Chords from an array

accordi[] holds the absolute notes of the chord (0..11), terminated by **99**. It is filled with **GeneraAccordi()** and played with **playAccordiNoBasePat()**:

```
GeneraAccordi(tonalita, tipo, 1, accordi, riga); // fills accordi[]  
playAccordiNoBasePat(starti, base, ottava, notastart, spostata, modo, playorpat, midichan);
```

The **modo** parameter determines how the chord is played:

modo	Playing style	modo	Playing style
1	notes overlapped for the whole duration	8	notes of random length
2	notes one after another, duration split	9	arpeggio
3	partial overlap	10	basso continuo
4	multiple partial overlap	11	basso continuo in triplets
5	one hit of half duration	12	odd pattern

modo	Playing style	modo	Playing style
6	two hits	13	random by thirds/fifths
7	random blocks of 1/8	14	arpeggio 0-1-0-2

With **modo = 0** the choice is random between 1 and 2.

- **playorpat**: **0** plays immediately, **1** generates a pattern instead of playing (useful for reworking it with effects).
- If **accordi[]** is empty the function does nothing.

10 Patterns: writing and generating them

The pattern is the raw material of every song: a sequence of slots, one per tick, each holding a scale degree and an octave (or a rest). Patterns can be written by hand, note by note, or generated by functions that apply different rules.

10.1 The basePat array

`basePat[i]` holds, for each tick, a scale degree (interval) and an octave. Its size is **BASEPATELE**: 512 on Mega and Due, 256 on other boards.

NOTE The automatic composer uses working areas high in the array (for example from 192 upwards): in hand-written songs it is better to use low indexes.

10.2 Reading and writing

Function	Description
<code>nota2int(nota_midi, i)</code>	writes the MIDI note into the pattern, moving it to the nearest degree of the scale
<code>int2int(grado, ottava, i)</code>	writes degree and octave directly (255, 99 = rest)
<code>int2nota(i)</code>	returns the MIDI note (255 if rest)
<code>duplicaPattern(starti, desti, n)</code>	copies n elements
<code>duplicaPatternInv(starti, desti, n)</code>	copies n elements in reverse order
<code>sumint / spostaPattern(starti, numnote, spostamento)</code>	shifts the notes by a number of degrees

10.3 Generators

All generators fill the pattern starting at **vettstart** for **base** ticks. **ottavastart** and **ottavaend** set the range of octaves used.

Function	Description
<code>creabasePat(fullp, base, npattern, ottavastart, ottavaend)</code>	random pattern of base ticks; fullp = true builds it; npattern = how many patterns to produce (to get just one in addition to the base, use 2)
<code>baseFullControl(vettstart, base, mmin, mmax, normalizza, modoblank, ottavastart, ottavaend)</code>	note changes at random intervals: mmin/mmax = minimum/maximum length (0 = random), normalizza = round to multiples, modoblank = rests (8.4)
<code>baseFixNote(numnote, base, vettstart, vettdest, tipocalcolo, ottavastart, ottavaend)</code>	numnote notes; tipocalcolo : 0 fully random, 1 copies the notes and changes the durations, 2 copies the durations and changes the notes
<code>baseRegularStep(vettstart, base, lenstep, ottavastart, ottavaend)</code>	notes at a regular step (lenstep 0 = random)
<code>basePesato(vettstart, base, lenstep, ottavastart, ottavaend)</code>	choices weighted on the complexp parameters (chapter 12)
<code>baseFibonacci(vettstart, base, modoblank, addval, ottavastart, ottavaend)</code>	built on the Fibonacci series
<code>baseFromAccordi(vettstart, base, lenstep, ottavastart, startacc)</code>	uses only the notes of the chords

10.4 Variations on an existing pattern

To avoid identical repetitions, an existing pattern can be varied slightly:

Function	Description
<code>notePlus(numnote, vettstart)</code>	variation on the existing pattern
<code>noteMinus(numnote, vettstart)</code>	variation on the existing pattern
<code>moduloMinDiff(starti, desti, base)</code>	small change to the longest note

11 Players and effects

Players are the functions that read a pattern and put the notes into **playList**. Effects are players that transform the pattern while reading it: they stretch it, reverse it, arpeggiate it. The same pattern can therefore be played in different ways on different tracks.

11.1 Common parameters

Parameter	Meaning
sti / starti	where the pattern starts in basePat
patlen / notepattern	how many ticks to read
sposnnota	raises the note by that number of scale degrees (0 = unchanged)
ltick	length in ticks used by the effect
midichan	MIDI channel / track tag (7.3)

11.2 Players

Function	Description
moduloBase(sti, notepattern, midichan)	plays the pattern; there are versions with modoaccordo , tipoaccordo , startvolume , diffvolume
moduloBasePlus(...)	like moduloBase , but ties the first note to the previous last one
moduloBaseDeviant(starti, notepattern, varlen, modoaccordo, tipoaccordo, midichan)	merges notes; varlen : 0 merges, 1 extends to the end, 2 random
moduloBaseSposta(sti, notepattern, sposnnota, midichan)	like moduloBase , shifted by degrees
moduloSingle / moduloRangeSingle / moduloRangeNota	single notes or notes spread across channels

11.3 Effects

Function	Description
moduloEspandi(sti, patlen, ltick, sposnnota, midichan)	stretches time: each tick lasts ltick
moduloContra(sti, patlen, divisore, sposnnota, midichan)	one note every divisore ticks
moduloBeatRev(sti, patlen, offset, sposnnota, midichan)	the bar backwards (offset raises/lowers)
moduloCounterpUniversal(sti, patlen, ltick, sposnnota, midichan)	counterpoint: offsets by ltick
moduloDivBattuta(starti, patlen, ltick, sposnnota, midichan)	normalises notes to length ltick
moduloCambioOttava(starti, ltick, midichan)	changes the octave at random
moduloTerze(sti, patlen, sposnnota, midichan)	shifts by thirds/fifths (0 = random $\pm 2/4$)
moduloTriads(starti, patlen, ltick, midichan)	triples the notes

Function	Description
<code>moduloOstinato(starti, patlen, ltick, tipoostinato, midichan)</code>	repeats the first note of each group (ostinato)
<code>moduloArp(starti, patlen, ltick, tipoarp, segnoflag, midichan)</code>	arpeggiator
<code>moduloTetraScala(ltick, ottava, midichan)</code>	scale 0-2-4-6 in four beats

Other effects – `moduloArpGlass`, `moduloMedia`, `moduloStrongBeat`, `moduloStrongPattern`, `moduloChangeLast`, `moduloPlayFirstLast`, `moduloNotaperPattern`, `multiAcc` – are documented in the comments at the top of each function in `LibFX.ino`.

11.4 Free modules

These modules do not read a pattern but generate notes directly:

`moduloTriangular`, `moduloSinPattern`, `moduloSinfree`, `moduloClusterKeyFull`, `moduloScalaRndKey`, `moduloMahler`, `moduloCluster`, `moduloClusterKey`, `moduloFullRandomNote`, `moduloKeyRandomNote`.

NOTE As in all modules, notes are queued with `playAdd(nota, midichan, volume, durata, inizio)` and play when their tick arrives.

12 The automatic composer (complexp)

Besides hand-written songs, the system includes an automatic composer that builds an entire song from a few “character” parameters: how aggressive, complex, modern or dark it should be.

12.1 The complexp structure

The `complexp` structure is defined in `H_Ard_NG50_Base.ino`; the initial values are set in `setDefaultSong()` (`Funzioni.ino`).

Field	Description	Default
<code>len</code>	length multiplier (1...); work in multiple blocks	2
<code>aggressivita</code>	aggressiveness, 0..10: probability of variations; 0 never, 10 always	5
<code>comp_orizzontale[0]</code>	horizontal complexity, 1..10: complexity of the function parameters	5
<code>comp_verticale[0]</code>	vertical complexity: number of active tracks 0..6; 10 = random	10
<code>modernita</code>	modernity: 0 classic tracks, 1 modern, 2 random per section, 3 random per section (0,1,2), 4 random for the whole piece (0,1 only)	3
<code>effetti[0]</code>	effects: 0 off, 1 always effect 1, 2 always effect 2, 3 both, 4 random	1
<code>accordi</code>	chords: active groups 1..3	1
<code>oscurita[0]</code>	darkness, 0..9: darkens the sound, acts on octave and length	5

NOTE The `[1]` elements are values processed/mapped by the system: do not write them by hand.

12.2 Effect on the song

`setDefaultSong()` derives the general characteristics of the song from the parameters:

- **BPM** = $20 + \text{random}(0..4) \times 5 + (\text{mapped oscurita}) \times 5$
- **Key** chosen according to aggressiveness: up to 3 among the most common ones, up to 6 also the modes, above that any key.

12.3 Sections and tracks

The full composer is `modeTrak(base1, sezione, lenpat, pos)` in `SongFunc.ino`; `genMainPatSong()` chooses the song structure.

Code	Section	Code	Section
<code>i</code>	intro	<code>r</code>	chorus
<code>s</code>	verse	<code>c</code>	recurring
<code>p</code>	bridge	<code>b</code>	break
<code>e</code>	special	<code>d</code>	coda
<code>l</code>	solo	<code>f</code>	ending

Track	Role
0	chords
1	melody
2	bass
3–6	accompaniment

NOTE The full composer is not called by `carica()`: the test songs are currently used (chapter 14).

13 Complete song example

This example puts together what has been covered so far: initialisation, key, a hand-written pattern with rests and an effect applied every four bars.

13.1 The code

Add the function, for example in a new file `MiaCanzone.ino`, and call it from `carica()` (chapter 14).

```
void MiaCanzone() {
  byte base1 = 16;                                // ticks per bar
  if (contick == 0) {                              // initialisation: once only, on the first tick
    PotAverage[9] = 30;                            // speed: bars per minute
                                                    // (1 tick = 60 / (BPM x 16) seconds)
    PotAverage[3] = 0;                            // key: 0 = C major
    setModeKeySong(PotValue(3));
    DatiVideo.baselen = base1;                    // display only: ticks per bar
    DatiVideo.songlen = 8;                        // display only: bars in the song
    for (int i = 0; i < base1; i++) nota2int(255, i); // start from an all-silent pattern...
    nota2int(60, 0); nota2int(60, 1);             // C C
    nota2int(62, 4);                              // D
    nota2int(64, 6); nota2int(64, 7);             // E E
  }
  if (contick % base1 == 0) {                     // at the start of every bar
    moduloBase(0, base1, 1);                      // play basePat[0..15] on channel 1
    // every fourth bar: the same pattern backwards on channel 2
    if (tickDivider(4, base1) == 3) moduloBeatRev(0, base1, 0, 0, 2);
  }
}
```

13.2 How it works

Part	Explanation
<code>contick == 0</code>	sets 30 BPM and C major, clears the pattern by filling it with rests and writes the five notes
<code>contick % base1 == 0</code>	on every bar (every 16 ticks) queues the pattern on channel 1
<code>tickDivider(4, base1) == 3</code>	one bar in four adds the same pattern backwards on channel 2

13.3 Result

At 30 BPM a bar lasts 2 seconds:

Note	Start	End
C	0 s	0.25 s
D	0.5 s	0.625 s
E	0.75 s	1 s
–	1 s	silence until the next bar

Replacing `moduloBase(0, base1, 1)` with `moduloBase(0, base1, 1, 1, 3)` turns every note into a major triad on channel 3.

13.4 Quick recipes

```
// change key halfway through the song (D minor)
PotAverage[3] = 14; setModeKeySong(PotValue(3));
// slow down
PotAverage[9] = 20;
// accent the strong beats
moduloStrongBeat(starti, patlen, countbeat, ltick, midichan);
```

14 Choosing the active song

In `CaricaF.ino` the `carica()` function calls a single song; the calls to the others are commented out. To activate a song, remove the `//` from its line and add it to the line of the currently active song.

Line in <code>carica()</code>	Description
<code>//PrimoNewMaggiore();</code>	pattern in a major key, changes key every 14 bars
<code>//PrimoNewMinore();</code>	as above, in minor
<code>//SingMajMin();</code>	single notes with major/minor chord, changes key
<code>//PlayAgain();</code>	four bars with chords and variations, then changes key
<code>//Song1();</code>	structured song (intro, verse, chorus, bridge); it uses <code>diffstop</code> , so with the two stop lines in <code>carica()</code> it stops on the first tick (see 7.5)
<code>MinTest();</code>	active song: generated pattern, expanded and repeated, base 32, 30 BPM

TIP For a new song: write the function (chapter 13) and call it instead of `MinTest()`.

15 Things to know

- The behaviours described have been verified in a PC simulation (ticks, notes, rests, chords, keys at startup, automatic start). Display, network, sensors and real-time timing have not been tested.
- Rests take no space in `playList`: they neither slow down nor fill the list.
- `BASEPATELE`, `PLAYELE` and `ENDELE` affect RAM usage: the Mega has only 8 KB in total.
- Functions marked “NON FUNZIONA” (does not work) in the original comments have not been modified.

Appendix A Troubleshooting

Symptom	Likely cause	Remedy
Nothing plays, but the serial port shows "Cmd: 144 ..."	debug mode is on: MIDI is sent as text	<code>debug = false</code> (4.1)
The song does not start at power-up	<code>accesso = false</code> , or KEYS_NONE without a command from the core unit	<code>accesso = true</code> or press start (1.1)
A key pressed at power-up has no effect	keys are ignored during the first 500 ms	press after startup (1.4)
Notes are out of time	the tick takes longer than the available interval	check <code>Tick max us</code> , exclude display/network, lower the BPM (4.3)
Missing notes, "0" is shown	<code>playlist</code> is full	raise <code>PLAYELE</code> above the peak (3.4)
Notes too short	<code>endList</code> is full: notes are switched off immediately	raise <code>ENDELE</code> (3.3)
Song1 stops immediately	it uses <code>diffstop</code> with the stop lines of <code>carica()</code>	see 7.5 and 14
Identical notes from different tracks merge	same <code>midichan</code>	use different channels (7.3)

Appendix B Glossary

Term	Definition
tick	smallest unit of time; lasts 60 / (BPM × 16) seconds
base1	number of ticks in one bar of the song
contick	tick counter since the start of the song
pattern	sequence of degrees and octaves in basePat , one slot per tick
degree	position of a note within the current scale; makes patterns transposable
rest	slot with interval 255 and octave 99
player	function that reads a pattern and queues the notes
effect	player that transforms the pattern while reading it
playList	list of notes waiting to be played
endList	list of sounding notes, to be switched off
midichan / tag	MIDI channel and track identifier
deferred end	stopping the song on the next tick via diffstop
core unit	external unit connected on Serial1 that sends text commands
complexp	“character” parameters of the automatic composer