```zig
    ///----------------------
    /// prog mdlRjson
    /// zig 0.13.0 dev
    ///----------------------
const std = @import("std");

// term

const term = @import("cursed");
// keyboard
const kbd = @import("cursed").kbd;

// panel
const pnl = @import("forms").pnl;
// button
const btn = @import("forms").btn;
// label
const lbl = @import("forms").lbl;
// flied
const fld = @import("forms").fld;
// line horizontal
const lnh = @import("forms").lnh;
// line vertival
const lnv = @import("forms").lnv;

// grid
const grd = @import("grid").grd;
// menu
const mnu = @import("menu").mnu;

// full delete for produc
const forms = @import("forms");


pub const allocatorJson = std.heap.page_allocator;


// tools utility
const utl = @import("utils");

//.............................//
// define Ctype JSON
const Ctype = enum { null, bool, integer, float, number_string, string, array, object, decimal_string };

//.............................//
// define BUTTON JSON
```

```
47: //.............................//
48:
49: const DEFBUTTON = struct { key: kbd, show: bool, check: bool, title: []const u8 };
50:
51: const Jbutton = enum { key, show, check, title };
52:
53: //.............................//
54: // define LABEL JSON
55: //.............................//
56:
57: const DEFLABEL = struct { name: []const u8, posx: usize, posy: usize, text: []const u8, title: bool };
58:
59: const Jlabel = enum { name, posx, posy, text, title };
60:
61: //.............................//
62: // define FIELD JSON
63: //.............................//
64:
65: pub const DEFFIELD = struct {
66:     name: []const u8,
67:     posx: usize,
68:     posy: usize,
69:     reftyp: forms.REFTYP,
70:     width: usize,
71:     scal: usize,
72:     requier: bool, // requier or FULL
73:     protect: bool, // only display
74:     edtcar: []const u8, // edtcar ex: monnaie
75:
76:     regex: []const u8, //contrÃ´le regex
77:     errmsg: []const u8, //message this field
78:
79:     help: []const u8, //help this field
80:
81:     text: []const u8,
82:     zwitch: bool, // CTRUE CFALSE
83:
84:     procfunc: []const u8, //name proc
85:
86:     proctask: []const u8, //name proc
87:
88:     progcall: []const u8, //name program
89:
90:     typecall: []const u8, //type SH APPTERM
91:
92:     parmcall: bool, // parm Yes/No
```

```
 93:
 94:     actif: bool,
 95: };
 96:
 97: const Jfield = enum { name, posx, posy, reftyp, width, scal, text, zwitch, requier, protect, edtcar, errmsg,
 98:      help, procfunc, proctask, progcall, typecall, parmcall, regex };
 99:
100: //.............................//
101: // define LINEV JSON
102: //.............................//
103:
104: const DEFLINEV = struct { name: []const u8, posx: usize, posy: usize, lng: usize, trace: forms.LINE };
105:
106: const Jlinev = enum { name, posx, posy, lng, trace };
107:
108: //.............................//
109: // define LINEH JSON
110: //.............................//
111:
112: const DEFLINEH = struct { name: []const u8, posx: usize, posy: usize, lng: usize, trace: forms.LINE };
113:
114: const Jlineh = enum { name, posx, posy, lng, trace };
115:
116: //.............................//
117: // define CELL JSON
118: //.............................//
119:
120: const DEFCELL = struct { text: []const u8, long: usize, reftyp: grd.REFTYP,
121:                          posy: usize, edtcar: []const u8, atrCell: term.ForegroundColor };
122:
123: const Jcell = enum {text, long, reftyp, posy, edtcar, atrcell };
124:
125: //.............................//
126: // define GRID JSON
127: //.............................//
128:
129: const ArgData = struct {
130:         buf: std.ArrayList([]const u8) = std.ArrayList([]const u8).init(allocatorJson),
131:     };
132:
133: const RGRID = struct { name: []const u8, posx: usize, posy: usize , pageRows: usize, separator: []const u8,
134:      cadre: grd.CADRE, cells: std.ArrayList(DEFCELL), data: std.MultiArrayList(ArgData)};
135:
136: const Jgrid = enum {name, posx, posy, pagerows, separator, cadre, cells, data};
137:
138:
```

```
139: //.............................//
140: // define MENU JSON
141: //.............................//
142:
143: const DEFMENU = struct { name: []const u8, posx: usize, posy: usize , cadre: mnu.CADRE, mnuvh: mnu.MNUVH,
144:                          xitems : [][] const u8};
145:
146: const Jmenu = enum {name, posx, posy, cadre, mnuvh, xitems};
147: const Jitem = enum {text};
148:
149:     pub fn initMenuDef(
150:         vname: []const u8,
151:         vposx: usize,
152:         vposy: usize,
153:         vcadre: mnu.CADRE,
154:         vmnuvh: mnu.MNUVH,
155:         vxitems : [][] const u8 )
156:         mnu.DEFMENU{
157:             const xmenu = mnu.DEFMENU{
158:             .name = vname,
159:             .posx = vposx,
160:             .posy = vposy,
161:             .cadre = vcadre,
162:             .mnuvh = vmnuvh,
163:             .xitems = vxitems
164:             };
165:         return xmenu;
166:     }
167: //.............................//
168: // define PANEL JSON
169: //.............................//
170: const RPANEL = struct { name: []const u8, posx: usize, posy: usize, lines: usize, cols: usize,
171:     cadre: forms.CADRE, title: []const u8,
172:     button: std.ArrayList(DEFBUTTON),
173:     label: std.ArrayList(DEFLABEL),
174:     field: std.ArrayList(DEFFIELD),
175:     linev: std.ArrayList(DEFLINEV),
176:     lineh: std.ArrayList(DEFLINEH) };
177:
178: const Jpanel = enum { name, posx, posy, lines, cols, cadre, title, button, label, field, linev, lineh };
179:
180: var NPANEL = std.ArrayList(RPANEL).init(allocatorJson);
181: var NGRID  = std.ArrayList(RGRID ).init(allocatorJson);
182: var NMENU  = std.ArrayList(DEFMENU ).init(allocatorJson);
183:
184: //.............................//
```

```zig
//  string return enum
//.............................//

fn strToEnum(comptime EnumTag: type, vtext: []const u8) EnumTag {
    inline for (@typeInfo(EnumTag).Enum.fields) |f| {
        if (std.mem.eql(u8, f.name, vtext)) return @field(EnumTag, f.name);
    }

    var buffer: [128]u8 = [_]u8{0} ** 128;
    const result = std.fmt.bufPrintZ(buffer[0..], "invalid Text {s} for strToEnum ", .{vtext}) catch unreachable;
    @panic(result);
}

//.............................//
// JSON
//.............................//

const T = struct {
    x: ?std.json.Value,
    var err: bool = false;

    pub fn init(self: std.json.Value) T {
        return T{ .x = self };
    }

    pub fn get(self: T, query: []const u8) T {
        err = false;

        if (self.x.?.object.get(query) == null) {
            //  std.debug.print("ERROR::{s}::{s}\n\n", .{"invalid",query});
            err = true;
            return T.init(self.x.?);
        }

        return T.init(self.x.?.object.get(query).?);
    }

    pub fn ctrlPack(self: T, Xtype: Ctype) bool {
        var out = std.ArrayList(u8).init(allocatorJson);
        defer out.deinit();

        switch (self.x.?) {
            .null => {
                if (Xtype != .null) return false;
            },

```

```
231:             .bool => {
232:                 if (Xtype != Ctype.bool) return false;
233:             },
234:
235:             .integer => {
236:                 if (Xtype != Ctype.integer) return false;
237:             },
238:
239:             .float => {
240:                 if (Xtype != Ctype.float) return false;
241:             },
242:
243:             .number_string => {
244:                 if (Xtype != Ctype.number_string) return false;
245:             },
246:
247:             .string => {
248:                 if (Xtype != Ctype.string) return false;
249:                 if (Xtype == Ctype.decimal_string)
250:                     return utl.isDecimalStr(std.fmt.allocPrint(allocatorJson, "{s}", .{self.x.?.string})
251:                         catch unreachable);
252:             },
253:
254:             .array => {
255:                 if (Xtype != Ctype.array) return false;
256:             },
257:
258:             .object => {
259:                 if (Xtype != Ctype.object) return false;
260:                 //try printPack(self,Xtype);
261:             },
262:         }
263:
264:         return true;
265:     }
266:
267:     pub fn index(self: T, i: usize) T {
268:         err = false;
269:         switch (self.x.?) {
270:             .array => {
271:                 if (i > self.x.?.array.items.len) {
272:                     std.debug.print("ERROR::{s}::\n\n", .{"index out of bounds"});
273:                     err = true;
274:                     return T.init(self.x.?);
275:                 }
276:             },
```

```
277:                 else => {
278:                     std.debug.print("ERROR::{s}:: {s}\n\n", .{ "Not array", @tagName(self.x.?) });
279:                     err = true;
280:                     return T.init(self.x.?);
281:                 },
282:             }
283:             return T.init(self.x.?.array.items[i]);
284:         }
285: };
286:
287: //.............................//
288: // DECODEUR
289: //.............................//
290:
291: pub fn jsonDecode(my_json: []const u8) !void {
292:     var val: T = undefined;
293:
294:     const parsed = try std.json.parseFromSlice(std.json.Value, allocatorJson, my_json, .{});
295:     defer parsed.deinit();
296:
297:     const json = T.init(parsed.value);
298:
299: //-----------------------------------------------------
300: // PANEL
301: //-----------------------------------------------------
302:
303:     val = json.get("PANEL");
304:
305:     const nbrPanel = val.x.?.array.items.len;
306:
307:     var p: usize = 0;
308:
309:     const Rpanel = std.enums.EnumIndexer(Jpanel);
310:
311:     const Rbutton = std.enums.EnumIndexer(Jbutton);
312:
313:     const Rlabel = std.enums.EnumIndexer(Jlabel);
314:
315:     const Rfield = std.enums.EnumIndexer(Jfield);
316:
317:     const Rlinev = std.enums.EnumIndexer(Jlinev);
318:
319:     const Rlineh = std.enums.EnumIndexer(Jlineh);
320:
321:     while (p < nbrPanel) : (p += 1) {
322:         var n: usize = 0; // index
```

```
323:
324:         NPANEL.append(RPANEL{ .name = "", .posx = 0, .posy = 0, .lines = 0, .cols = 0,
325:             .cadre = forms.CADRE.line0, .title = "",
326:             .button = std.ArrayList(DEFBUTTON).init(allocatorJson),
327:             .label = std.ArrayList(DEFLABEL).init(allocatorJson),
328:             .field = std.ArrayList(DEFFIELD).init(allocatorJson),
329:             .linev = std.ArrayList(DEFLINEV).init(allocatorJson),
330:             .lineh = std.ArrayList(DEFLINEH).init(allocatorJson) }) catch unreachable;
331:
332:         while (n < Rpanel.count) : (n += 1) {
333:             var v: usize = 0; // index
334:             var y: usize = 0; // array len
335:             var z: usize = 0; // compteur
336:             var b: usize = 0; // button
337:             var l: usize = 0; // label
338:             var f: usize = 0; // field
339:             var vx: usize = 0; // line vertical
340:             var hx: usize = 0; // line horizontal
341:
342:             switch (Rpanel.keyForIndex(n)) {
343:                 Jpanel.name => {
344:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
345:                     if (T.err) break;
346:
347:                     if (val.ctrlPack(Ctype.string))
348:                         NPANEL.items[p].name = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
349:                     else
350:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  Panel err_Field :{s}\n",
351:                             .{@tagName(Rpanel.keyForIndex(n))}));
352:                 },
353:                 Jpanel.posx => {
354:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
355:
356:                     if (val.ctrlPack(Ctype.integer))
357:                         NPANEL.items[p].posx = @intCast(val.x.?.integer)
358:                     else
359:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
360:                             .{@tagName(Rpanel.keyForIndex(n))}));
361:                 },
362:                 Jpanel.posy => {
363:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
364:
365:                     if (val.ctrlPack(Ctype.integer))
366:                         NPANEL.items[p].posy = @intCast(val.x.?.integer)
367:                     else
368:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
```

```
369:                         .{@tagName(Rpanel.keyForIndex(n))}));
370:                 },
371:                 Jpanel.lines => {
372:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
373:
374:                     if (val.ctrlPack(Ctype.integer))
375:                         NPANEL.items[p].lines = @intCast(val.x.?.integer)
376:                     else
377:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
378:                             .{@tagName(Rpanel.keyForIndex(n))}));
379:                 },
380:                 Jpanel.cols => {
381:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
382:
383:                     if (val.ctrlPack(Ctype.integer))
384:                         NPANEL.items[p].cols = @intCast(val.x.?.integer)
385:                     else
386:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
387:                             .{@tagName(Rpanel.keyForIndex(n))}));
388:                 },
389:                 Jpanel.cadre => {
390:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
391:
392:                     if (val.ctrlPack(Ctype.string)) {
393:                         NPANEL.items[p].cadre = strToEnum(forms.CADRE, val.x.?.string);
394:                     } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
395:                         .{@tagName(Rpanel.keyForIndex(n))}));
396:                 },
397:                 Jpanel.title => {
398:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
399:
400:                     if (val.ctrlPack(Ctype.string))
401:                         NPANEL.items[p].title = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
402:                     else
403:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
404:                             .{@tagName(Rpanel.keyForIndex(n))}));
405:                 },
406:                 //=============================================================================
407:                 // BUTTON
408:                 //=============================================================================
409:                 Jpanel.button => {
410:                     val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
411:                     if (T.err) break;
412:
413:                     var bt: DEFBUTTON = undefined;
414:                     y = val.x.?.array.items.len;
```

```
415:                     z = 0;
416:                     b = 0;
417:
418:                     while (z < y) : (z += 1) {
419:                         v = 0;
420:                         while (v < Rbutton.count) : (v += 1) {
421:                             val = json.get("PANEL").index(p).get("button").index(b)
422:                                 .get(@tagName(Rbutton.keyForIndex(v)));
423:
424:                             switch (Rbutton.keyForIndex(v)) {
425:                                 Jbutton.key => {
426:                                     if (val.ctrlPack(Ctype.string)) {
427:                                         bt.key = strToEnum(kbd, val.x.?.string);
428:                                     } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
429:                                         .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rbutton.keyForIndex(v)) }));
430:                                 },
431:                                 Jbutton.show => {
432:                                     if (val.ctrlPack(Ctype.bool))
433:                                         bt.show = val.x.?.bool
434:                                     else
435:                                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
436:                                             .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rbutton.keyForIndex(v)) }));
437:                                 },
438:                                 Jbutton.check => {
439:                                     if (val.ctrlPack(Ctype.bool))
440:                                         bt.check = val.x.?.bool
441:                                     else
442:                                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
443:                                             .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rbutton.keyForIndex(v)) }));
444:                                 },
445:                                 Jbutton.title => {
446:                                     if (val.ctrlPack(Ctype.string))
447:                                         bt.title = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
448:                                     else
449:                                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
450:                                             .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rbutton.keyForIndex(v)) }));
451:
452:                                     NPANEL.items[p].button.append(bt) catch unreachable;
453:                                 },
454:                             }
455:                         }
456:                         b += 1;
457:                     }
458:                 },
459:                 //=============================================================================
460:                 // LABEL
```

```
461:                //=============================================================================
462:
463:                Jpanel.label => {
464:                    val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
465:                    if (T.err) break;
466:
467:                    var lb: DEFLABEL = undefined;
468:                    y = val.x.?.array.items.len;
469:                    z = 0;
470:                    l = 0;
471:                    while (z < y) : (z += 1) {
472:                        v = 0;
473:                        while (v < Rlabel.count) : (v += 1) {
474:                            val = json.get("PANEL").index(p).get("label").index(l)
475:                                .get(@tagName(Rlabel.keyForIndex(v)));
476:
477:                            switch (Rlabel.keyForIndex(v)) {
478:                                Jlabel.name => {
479:                                    if (val.ctrlPack(Ctype.string))
480:                                        lb.name = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
481:                                    else
482:                                        @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
483:                                            .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlabel.keyForIndex(v)) }));
484:                                },
485:                                Jlabel.posx => {
486:                                    if (val.ctrlPack(Ctype.integer)) {
487:                                        lb.posx = @intCast(val.x.?.integer);
488:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
489:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlabel.keyForIndex(v)) }));
490:                                },
491:                                Jlabel.posy => {
492:                                    if (val.ctrlPack(Ctype.integer)) {
493:                                        lb.posy = @intCast(val.x.?.integer);
494:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
495:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlabel.keyForIndex(v)) }));
496:                                },
497:                                Jlabel.text => {
498:                                    if (val.ctrlPack(Ctype.string))
499:                                        lb.text = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
500:                                    else
501:                                        @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
502:                                            .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlabel.keyForIndex(v)) }));
503:                                },
504:                                Jlabel.title => {
505:                                    if (val.ctrlPack(Ctype.bool))
506:                                        lb.title = val.x.?.bool
```

```
507:                                    else
508:                                        @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
509:                                            .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlabel.keyForIndex(v)) }));
510:
511:                                    NPANEL.items[p].label.append(lb) catch unreachable;
512:                                },
513:                            }
514:                        }
515:
516:                        l += 1;
517:                    }
518:                },
519:                //=============================================================================
520:                // FIELD
521:                //=============================================================================
522:
523:                Jpanel.field => {
524:                    val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
525:                    if (T.err) break;
526:
527:                    var sreftyp: []const u8 = undefined;
528:
529:                    var lf: DEFFIELD = undefined;
530:                    y = val.x.?.array.items.len;
531:                    if (y == 0) break;
532:
533:                    z = 0;
534:                    f = 0;
535:                    while (z < y) : (z += 1) {
536:                        v = 0;
537:                        while (v < Rfield.count) : (v += 1) {
538:                            val = json.get("PANEL").index(p).get("field").index(f)
539:                                .get(@tagName(Rfield.keyForIndex(v)));
540:
541:                            switch (Rfield.keyForIndex(v)) {
542:                                Jfield.name => {
543:                                    if (val.ctrlPack(Ctype.string))
544:                                        lf.name = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
545:                                    else
546:                                        @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
547:                                            .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
548:                                },
549:
550:                                Jfield.posx => {
551:                                    if (val.ctrlPack(Ctype.integer)) {
552:                                        lf.posx = @intCast(val.x.?.integer);
```

```
553:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
554:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
555:                                },
556:
557:                                Jfield.posy => {
558:                                    if (val.ctrlPack(Ctype.integer)) {
559:                                        lf.posy = @intCast(val.x.?.integer);
560:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
561:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
562:                                },
563:
564:                                Jfield.reftyp => {
565:                                    if (val.ctrlPack(Ctype.string)) {
566:                                        sreftyp = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
567:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
568:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
569:
570:                                    lf.reftyp = strToEnum(forms.REFTYP, sreftyp);
571:                                },
572:
573:                                Jfield.width => {
574:                                    if (val.ctrlPack(Ctype.integer)) {
575:                                        lf.width = @intCast(val.x.?.integer);
576:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
577:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
578:                                },
579:
580:                                Jfield.scal => {
581:                                    if (val.ctrlPack(Ctype.integer)) {
582:                                        lf.scal = @intCast(val.x.?.integer);
583:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
584:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
585:                                },
586:
587:                                Jfield.text => {
588:                                    lf.text = "";
589:                                },
590:
591:                                Jfield.zwitch => {
592:                                    lf.zwitch = false;
593:                                },
594:
595:                                Jfield.requier => {
596:                                    if (val.ctrlPack(Ctype.bool)) {
597:                                        lf.requier = val.x.?.bool;
598:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
```

```
599:                                    .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
600:                            },
601:
602:                            Jfield.protect => {
603:                                if (val.ctrlPack(Ctype.bool)) {
604:                                    lf.protect = val.x.?.bool;
605:                                } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
606:                                    .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
607:                            },
608:
609:                            Jfield.edtcar => {
610:                                if (val.ctrlPack(Ctype.string)) {
611:                                    lf.edtcar = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
612:                                } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
613:                                    .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
614:                            },
615:
616:                            Jfield.errmsg => {
617:                                if (val.ctrlPack(Ctype.string)) {
618:                                    lf.errmsg = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
619:                                } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
620:                                    .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
621:                            },
622:
623:                            Jfield.help => {
624:                                if (val.ctrlPack(Ctype.string)) {
625:                                    lf.help = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
626:                                } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
627:                                    .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
628:                            },
629:
630:                            Jfield.procfunc => {
631:                                if (val.ctrlPack(Ctype.string)) {
632:                                    lf.procfunc = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
633:                                } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
634:                                    .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
635:                            },
636:
637:                            Jfield.proctask => {
638:                                if (val.ctrlPack(Ctype.string)) {
639:                                    lf.proctask = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
640:                                } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
641:                                    .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
642:                            },
643:
644:                            Jfield.progcall => {
```

```
645:                                 if (val.ctrlPack(Ctype.string)) {
646:                                     lf.progcall = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
647:                                 } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
648:                                     .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
649:                             },
650:
651:                             Jfield.typecall => {
652:                                 if (val.ctrlPack(Ctype.string)) {
653:                                     lf.typecall = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string});
654:                                 } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
655:                                     .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
656:                             },
657:
658:                             Jfield.parmcall => {
659:                                 if (val.ctrlPack(Ctype.bool)) {
660:                                     lf.parmcall = val.x.?.bool;
661:                                 } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
662:                                     .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rfield.keyForIndex(v)) }));
663:                             },
664:
665:                             Jfield.regex => {
666:                                 lf.regex = "";
667:
668:                                 NPANEL.items[p].field.append(lf) catch unreachable;
669:                             },
670:                         }
671:                     }
672:                     f += 1;
673:                 }
674:             },
675:             //=============================================================================
676:             // LINEV
677:             //=============================================================================
678:
679:             Jpanel.linev => {
680:                 val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
681:                 if (T.err) break;
682:
683:                 var lv: DEFLINEV = undefined;
684:                 y = val.x.?.array.items.len;
685:                 z = 0;
686:                 vx = 0;
687:                 while (z < y) : (z += 1) {
688:                     v = 0;
689:                     while (v < Rlinev.count) : (v += 1) {
690:                         val = json.get("PANEL").index(p).get("linev").index(vx)
```

```
691:                                .get(@tagName(Rlinev.keyForIndex(v)));
692:
693:                            switch (Rlinev.keyForIndex(v)) {
694:                                Jlinev.name => {
695:                                    if (val.ctrlPack(Ctype.string))
696:                                        lv.name = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
697:                                    else
698:                                        @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
699:                                            .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlinev.keyForIndex(v)) }));
700:                                },
701:                                Jlinev.posx => {
702:                                    if (val.ctrlPack(Ctype.integer)) {
703:                                        lv.posx = @intCast(val.x.?.integer);
704:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
705:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlinev.keyForIndex(v)) }));
706:                                },
707:                                Jlinev.posy => {
708:                                    if (val.ctrlPack(Ctype.integer)) {
709:                                        lv.posy = @intCast(val.x.?.integer);
710:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
711:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlinev.keyForIndex(v)) }));
712:                                },
713:                                Jlinev.lng => {
714:                                    if (val.ctrlPack(Ctype.integer)) {
715:                                        lv.lng = @intCast(val.x.?.integer);
716:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
717:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlinev.keyForIndex(v)) }));
718:                                },
719:                                Jlinev.trace => {
720:                                    if (val.ctrlPack(Ctype.string)) {
721:                                        lv.trace = strToEnum(forms.LINE, val.x.?.string);
722:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
723:                                        .{@tagName(Rlinev.keyForIndex(n))}));
724:
725:                                    NPANEL.items[p].linev.append(lv) catch unreachable;
726:                                },
727:                            }
728:                        }
729:
730:                        vx += 1;
731:                    }
732:                },
733:                //=============================================================================
734:                // LINEH
735:                //=============================================================================
736:
```

```
737:                Jpanel.lineh => {
738:                    val = json.get("PANEL").index(p).get(@tagName(Rpanel.keyForIndex(n)));
739:                    if (T.err) break;
740:
741:                    var lh: DEFLINEH = undefined;
742:                    y = val.x.?.array.items.len;
743:                    z = 0;
744:                    hx = 0;
745:                    while (z < y) : (z += 1) {
746:                        v = 0;
747:                        while (v < Rlineh.count) : (v += 1) {
748:                            val = json.get("PANEL").index(p).get("lineh").index(hx)
749:                                .get(@tagName(Rlineh.keyForIndex(v)));
750:
751:                            switch (Rlineh.keyForIndex(v)) {
752:                                Jlineh.name => {
753:                                    if (val.ctrlPack(Ctype.string))
754:                                        lh.name = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
755:                                    else
756:                                        @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
757:                                            .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlineh.keyForIndex(v)) }));
758:                                },
759:                                Jlineh.posx => {
760:                                    if (val.ctrlPack(Ctype.integer)) {
761:                                        lh.posx = @intCast(val.x.?.integer);
762:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
763:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlineh.keyForIndex(v)) }));
764:                                },
765:                                Jlineh.posy => {
766:                                    if (val.ctrlPack(Ctype.integer)) {
767:                                        lh.posy = @intCast(val.x.?.integer);
768:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
769:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlineh.keyForIndex(v)) }));
770:                                },
771:                                Jlineh.lng => {
772:                                    if (val.ctrlPack(Ctype.integer)) {
773:                                        lh.lng = @intCast(val.x.?.integer);
774:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
775:                                        .{ @tagName(Rpanel.keyForIndex(n)), @tagName(Rlineh.keyForIndex(v)) }));
776:                                },
777:                                Jlineh.trace => {
778:                                    if (val.ctrlPack(Ctype.string)) {
779:                                        lh.trace = strToEnum(forms.LINE, val.x.?.string);
780:                                    } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
781:                                        .{@tagName(Rlineh.keyForIndex(n))}));
782:
```

```
783:                                     NPANEL.items[p].lineh.append(lh) catch unreachable;
784:                                 },
785:                             }
786:                         }
787:
788:                         hx += 1;
789:                     }
790:                 },
791:             }
792:         }
793:     }
794:
795: //--------------------------------------------------
796: // GRID
797: //--------------------------------------------------
798:
799:     val = json.get("GRID");
800:     if (!T.err)  {
801:     const nbrGrid = val.x.?.array.items.len;
802:     var g: usize = 0;
803:     const Rcell = std.enums.EnumIndexer(Jcell);
804:     const Rgrid = std.enums.EnumIndexer(Jgrid);
805:
806:     while (g < nbrGrid) : (g += 1) {
807:
808:         NGRID.append(RGRID{.name = "", .posx = 0, .posy = 0, .pageRows = 0 ,
809:             .separator = " ", .cadre = grd.CADRE.line1,
810:             .cells = std.ArrayList(DEFCELL).init(allocatorJson),
811:             .data  = std.MultiArrayList(ArgData){}  }
812:         ) catch unreachable;
813:
814:         var n: usize = 0; // index
815:     while (n < Rgrid.count) : (n += 1) {
816:         var v: usize = 0; // index
817:         var y: usize = 0; // array len
818:         var z: usize = 0; // compteur
819:         var c: usize = 0; // cell
820:         switch (Rgrid.keyForIndex(n)) {
821:             Jgrid.name => {
822:                     val = json.get("GRID").index(g).get(@tagName(Rgrid.keyForIndex(n)));
823:                     if (T.err) break;
824:
825:                     if (val.ctrlPack(Ctype.string))
826:                         NGRID.items[g].name = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
827:                     else
828:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  Panel err_Field :{s}\n",
```

```
829:                                .{@tagName(Rgrid.keyForIndex(n))}));
830:             },
831:             Jgrid.posx => {
832:                 val = json.get("GRID").index(g).get(@tagName(Rgrid.keyForIndex(n)));
833:
834:                 if (val.ctrlPack(Ctype.integer))
835:                     NGRID.items[g].posx = @intCast(val.x.?.integer)
836:                 else
837:                     @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
838:                         .{@tagName(Rgrid.keyForIndex(n))}));
839:             },
840:             Jgrid.posy => {
841:                 val = json.get("GRID").index(g).get(@tagName(Rgrid.keyForIndex(n)));
842:
843:                 if (val.ctrlPack(Ctype.integer))
844:                     NGRID.items[g].posy = @intCast(val.x.?.integer)
845:                 else
846:                     @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
847:                         .{@tagName(Rgrid.keyForIndex(n))}));
848:             },
849:             Jgrid.pagerows => {
850:                 val = json.get("GRID").index(g).get(@tagName(Rgrid.keyForIndex(n)));
851:
852:                 if (val.ctrlPack(Ctype.integer))
853:                     NGRID.items[g].pageRows = @intCast(val.x.?.integer)
854:                 else
855:                     @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
856:                         .{@tagName(Rgrid.keyForIndex(n))}));
857:             },
858:             Jgrid.separator => {
859:                     val = json.get("GRID").index(g).get(@tagName(Rgrid.keyForIndex(n)));
860:                     if (T.err) break;
861:
862:                     if (val.ctrlPack(Ctype.string))
863:                         NGRID.items[g].separator = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
864:                     else
865:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  Panel err_Field :{s}\n",
866:                             .{@tagName(Rgrid.keyForIndex(n))}));
867:             },
868:             Jgrid.cadre => {
869:                 val = json.get("GRID").index(g).get(@tagName(Rgrid.keyForIndex(n)));
870:
871:                 if (val.ctrlPack(Ctype.string)) {
872:                     NGRID.items[g].cadre = strToEnum(grd.CADRE, val.x.?.string);
873:                 } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
874:                     .{@tagName(Rgrid.keyForIndex(n))}));
```

```
875:                },
876:                Jgrid.cells => {
877:                    val = json.get("GRID").index(g).get(@tagName(Rgrid.keyForIndex(n)));
878:                    if (T.err) break;
879:
880:                    var cl: DEFCELL = undefined;
881:                    var sreftyp: []const u8 = undefined;
882:                    var satrcell: []const u8 = undefined;
883:                    y = val.x.?.array.items.len;
884:                    z = 0;
885:                    c = 0;
886:                    while (z < y) : (z += 1) {
887:                        v = 0;
888:                        while (v < Rcell.count) : (v += 1  ) {
889:                            val = json.get("GRID").index(g).get("cells").index(c)
890:                                    .get(@tagName(Rcell.keyForIndex(v)));
891:
892:                            switch (Rcell.keyForIndex(v)) {
893:                                Jcell.text => {
894:                                    if (val.ctrlPack(Ctype.string))
895:                                        cl.text = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
896:                                    else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
897:                                        .{ @tagName(Rgrid.keyForIndex(g)), @tagName(Rgrid.keyForIndex(v)) }));
898:                                },
899:                                Jcell.long => {
900:                                    if (val.ctrlPack(Ctype.integer))
901:                                        cl.long = @intCast(val.x.?.integer)
902:                                    else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
903:                                        .{ @tagName(Rgrid.keyForIndex(g)), @tagName(Rgrid.keyForIndex(v)) }));
904:
905:                                },
906:                                Jcell.reftyp => {
907:                                    if (val.ctrlPack(Ctype.string))
908:                                        sreftyp = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
909:                                    else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
910:                                        .{ @tagName(Rgrid.keyForIndex(g)), @tagName(Rgrid.keyForIndex(v)) }));
911:                                    cl.reftyp = strToEnum(grd.REFTYP, sreftyp);
912:                                },
913:                                Jcell.posy => {
914:                                    if (val.ctrlPack(Ctype.integer))
915:                                    cl.posy = @intCast(val.x.?.integer)
916:                                    else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
917:                                        .{ @tagName(Rgrid.keyForIndex(g)), @tagName(Rgrid.keyForIndex(v)) }));
918:
919:                                },
920:                                Jcell.edtcar => {
```

```
921:                                    if (val.ctrlPack(Ctype.string))
922:                                    cl.edtcar = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
923:                                    else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
924:                                        .{ @tagName(Rgrid.keyForIndex(g)), @tagName(Rgrid.keyForIndex(v)) }));
925:                                },
926:                                Jcell.atrcell => {
927:                                    if (val.ctrlPack(Ctype.string))
928:                                        satrcell = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
929:                                    else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
930:                                        .{ @tagName(Rgrid.keyForIndex(g)), @tagName(Rgrid.keyForIndex(v)) }));
931:                                    cl.atrCell = strToEnum(term.ForegroundColor, satrcell);
932:                                    NGRID.items[g].cells.append(cl) catch |err| {@panic(@errorName(err));};
933:                                }
934:                            }
935:                        }
936:                        c += 1 ;
937:                    }
938:                },
939:                Jgrid.data => {}
940:            }
941:        } // Rgrid
942:        } //nbrGrid
943:        } // Terr
944: //------------------------------------------------------
945: // MENU
946: //------------------------------------------------------
947:
948:        val = json.get("MENU");
949:        if (!T.err)  {
950:        const nbrMenu = val.x.?.array.items.len;
951:        var m: usize = 0;
952:        const Ritem = std.enums.EnumIndexer(Jitem);
953:        const Rmenu = std.enums.EnumIndexer(Jmenu);
954:
955:        while (m < nbrMenu) : (m += 1) {
956:
957:            NMENU.append(DEFMENU{.name = "", .posx = 0, .posy = 0,
958:                 .cadre = mnu.CADRE.line1, .mnuvh = mnu.MNUVH.vertical ,
959:                 .xitems  = undefined,
960:                 }
961:
962:            ) catch unreachable;
963:
964:            var n: usize = 0; // index
965:        while (n < Rmenu.count) : (n += 1) {
966:            var v: usize = 0; // index
```

```
 967:         var y: usize = 0; // array len
 968:         var z: usize = 0; // compteur
 969:         var c: usize = 0; // cell
 970:         switch (Rmenu.keyForIndex(n)) {
 971:             Jmenu.name => {
 972:                     val = json.get("MENU").index(m).get(@tagName(Rmenu.keyForIndex(n)));
 973:                     if (T.err) break;
 974:
 975:                     if (val.ctrlPack(Ctype.string))
 976:                         NMENU.items[m].name = try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string})
 977:                     else
 978:                         @panic(try std.fmt.allocPrint(allocatorJson, "Json  Panel err_Field :{s}\n",
 979:                             .{@tagName(Rmenu.keyForIndex(n))}));
 980:             },
 981:             Jmenu.posx => {
 982:                 val = json.get("MENU").index(m).get(@tagName(Rmenu.keyForIndex(n)));
 983:
 984:                 if (val.ctrlPack(Ctype.integer))
 985:                     NMENU.items[m].posx = @intCast(val.x.?.integer)
 986:                 else
 987:                     @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
 988:                         .{@tagName(Rmenu.keyForIndex(n))}));
 989:             },
 990:             Jmenu.posy => {
 991:                 val = json.get("MENU").index(m).get(@tagName(Rmenu.keyForIndex(n)));
 992:
 993:                 if (val.ctrlPack(Ctype.integer))
 994:                     NMENU.items[m].posy = @intCast(val.x.?.integer)
 995:                 else
 996:                     @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
 997:                         .{@tagName(Rmenu.keyForIndex(n))}));
 998:             },
 999:             Jmenu.cadre => {
1000:                 val = json.get("MENU").index(m).get(@tagName(Rmenu.keyForIndex(n)));
1001:
1002:                 if (val.ctrlPack(Ctype.string)) {
1003:                     NMENU.items[m].cadre = strToEnum(mnu.CADRE, val.x.?.string);
1004:                 } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
1005:                     .{@tagName(Rmenu.keyForIndex(n))}));
1006:             },
1007:             Jmenu.mnuvh => {
1008:                 val = json.get("MENU").index(m).get(@tagName(Rmenu.keyForIndex(n)));
1009:
1010:                 if (val.ctrlPack(Ctype.string)) {
1011:                     NMENU.items[m].mnuvh = strToEnum(mnu.MNUVH, val.x.?.string);
1012:                 } else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}\n",
```

```
1013:                         .{@tagName(Rmenu.keyForIndex(n))}));
1014:                 },
1015:                 Jmenu.xitems => {
1016:                     val = json.get("MENU").index(m).get(@tagName(Rmenu.keyForIndex(n)));
1017:                     if (T.err) break;
1018:                     var xmenu =  std.ArrayList([]const u8).init(allocatorJson);
1019:                     defer xmenu.clearAndFree();
1020:                     y = val.x.?.array.items.len;
1021:                     z = 0;
1022:                     c = 0;
1023:                     while (z < y) : (z += 1) {
1024:                         v = 0;
1025:                         while (v < Ritem.count) : (v += 1  ) {
1026:                             val = json.get("MENU").index(m).get("xitems").index(c)
1027:                                     .get(@tagName(Ritem.keyForIndex(v)));
1028:
1029:                             switch (Ritem.keyForIndex(v)) {
1030:                                 Jitem.text => {
1031:                                     if (val.ctrlPack(Ctype.string))
1032:                                     try xmenu.append(try std.fmt.allocPrint(allocatorJson, "{s}", .{val.x.?.string}))
1033:                                     else @panic(try std.fmt.allocPrint(allocatorJson, "Json  err_Field :{s}.{s}\n",
1034:                                         .{ @tagName(Rmenu.keyForIndex(m)), @tagName(Rmenu.keyForIndex(v)) }));
1035:                                 }
1036:                             }
1037:                         }
1038:                         c += 1 ;
1039:                     }
1040:
1041:                     NMENU.items[m].xitems = try allocatorJson.alloc([]const u8, y);
1042:                     for (xmenu.items, 0.. ) |_,idx | {
1043:                         NMENU.items[m].xitems[idx]  = xmenu.items[idx];
1044:                     }
1045:                 },
1046:             }
1047:         } // nbrMenu
1048:         } // Rmenu
1049:         } // Terr
1050:
1051: }
1052:
1053: //.............................//
1054: // Main function
1055: //.............................//
1056: pub fn RstJson(XPANEL: *std.ArrayList(pnl.PANEL),
1057:             XGRID: *std.ArrayList(grd.GRID),
1058:             XMENU: *std.ArrayList(mnu.DEFMENU),
```

```
1059:         nameJson: []const u8) !void {
1060:
1061:     const cDIR = std.fs.cwd().openDir("dspf", .{}) catch |err| {
1062:         @panic(try std.fmt.allocPrint(allocatorJson, "err Open.{any}\n", .{err}));
1063:     };
1064:
1065:     var my_file = cDIR.openFile(nameJson, .{}) catch |err| {
1066:         @panic(try std.fmt.allocPrint(allocatorJson, "err Open.{any}\n", .{err}));
1067:     };
1068:     defer my_file.close();
1069:
1070:     const file_size = try my_file.getEndPos();
1071:     var buffer: []u8 = allocatorJson.alloc(u8, file_size) catch unreachable;
1072:
1073:     _ = try my_file.read(buffer[0..buffer.len]);
1074:
1075:     jsonDecode(buffer) catch |err| {
1076:         @panic(try std.fmt.allocPrint(allocatorJson, "err JsonDecode.{any}\n", .{err}));
1077:     };
1078:
1079:     XPANEL.clearAndFree();
1080:
1081:     for (NPANEL.items, 0..) |pnlx, idx| {
1082:         var vPanel: pnl.PANEL = undefined;
1083:         vPanel = pnl.initPanel(NPANEL.items[idx].name, NPANEL.items[idx].posx, NPANEL.items[idx].posy,
1084:             NPANEL.items[idx].lines, NPANEL.items[idx].cols, NPANEL.items[idx].cadre, NPANEL.items[idx].title);
1085:
1086:         for (pnlx.button.items) |p| {
1087:             var vButton: btn.BUTTON = undefined;
1088:
1089:             vButton = btn.newButton(p.key, p.show, p.check, p.title);
1090:
1091:             vPanel.button.append(vButton) catch |err| {
1092:                 @panic(@errorName(err));
1093:             };
1094:         }
1095:
1096:         for (pnlx.label.items) |p| {
1097:             var vLabel: lbl.LABEL = undefined;
1098:
1099:             if (p.title) vLabel = lbl.newTitle(p.name, p.posx, p.posy, p.text)
1100:             else vLabel = lbl.newLabel(p.name, p.posx, p.posy, p.text);
1101:
1102:             vPanel.label.append(vLabel) catch |err| {
1103:                 @panic(@errorName(err));
1104:             };
```

```
1105:         }
1106:
1107:         for (pnlx.field.items) |p| {
1108:             var vField: fld.FIELD = undefined;
1109:             switch (p.reftyp) {
1110:                 forms.REFTYP.TEXT_FREE => {
1111:                     vField = fld.newFieldTextFree(
1112:                         p.name,
1113:                         p.posx,
1114:                         p.posy,
1115:                         p.width,
1116:                         p.text,
1117:                         p.requier,
1118:                         p.errmsg,
1119:                         p.help,
1120:                         p.regex,
1121:                     );
1122:                     vField.proctask = p.proctask;
1123:                     vField.progcall = p.progcall;
1124:                     vField.typecall = p.typecall;
1125:                     vField.parmcall = p.parmcall;
1126:                     vField.protect = p.protect;
1127:                 },
1128:
1129:                 forms.REFTYP.TEXT_FULL => {
1130:                     vField = fld.newFieldTextFull(
1131:                         p.name,
1132:                         p.posx,
1133:                         p.posy,
1134:                         p.width,
1135:                         p.text,
1136:                         p.requier,
1137:                         p.errmsg,
1138:                         p.help,
1139:                         p.regex,
1140:                     );
1141:                     vField.proctask = p.proctask;
1142:                     vField.progcall = p.progcall;
1143:                     vField.typecall = p.typecall;
1144:                     vField.parmcall = p.parmcall;
1145:                     vField.protect = p.protect;
1146:                 },
1147:
1148:                 forms.REFTYP.ALPHA => {
1149:                     vField = fld.newFieldAlpha(
1150:                         p.name,
```

```
1151:                           p.posx,
1152:                           p.posy,
1153:                           p.width,
1154:                           p.text,
1155:                           p.requier,
1156:                           p.errmsg,
1157:                           p.help,
1158:                           p.regex,
1159:                       );
1160:                       vField.proctask = p.proctask;
1161:                       vField.progcall = p.progcall;
1162:                       vField.typecall = p.typecall;
1163:                       vField.parmcall = p.parmcall;
1164:                       vField.protect = p.protect;
1165:                   },
1166:
1167:                   forms.REFTYP.ALPHA_UPPER => {
1168:                       vField = fld.newFieldAlphaUpper(
1169:                           p.name,
1170:                           p.posx,
1171:                           p.posy,
1172:                           p.width,
1173:                           p.text,
1174:                           p.requier,
1175:                           p.errmsg,
1176:                           p.help,
1177:                           p.regex,
1178:                       );
1179:                       vField.proctask = p.proctask;
1180:                       vField.progcall = p.progcall;
1181:                       vField.typecall = p.typecall;
1182:                       vField.parmcall = p.parmcall;
1183:                       vField.protect = p.protect;
1184:                   },
1185:
1186:                   forms.REFTYP.ALPHA_NUMERIC => {
1187:                       vField = fld.newFieldAlphaNumeric(
1188:                           p.name,
1189:                           p.posx,
1190:                           p.posy,
1191:                           p.width,
1192:                           p.text,
1193:                           p.requier,
1194:                           p.errmsg,
1195:                           p.help,
1196:                           p.regex,
```

```
1197:                     );
1198:                     vField.proctask = p.proctask;
1199:                     vField.progcall = p.progcall;
1200:                     vField.typecall = p.typecall;
1201:                     vField.parmcall = p.parmcall;
1202:                     vField.protect = p.protect;
1203:                 },
1204:
1205:                 forms.REFTYP.ALPHA_NUMERIC_UPPER => {
1206:                     vField = fld.newFieldAlphaNumericUpper(
1207:                         p.name,
1208:                         p.posx,
1209:                         p.posy,
1210:                         p.width,
1211:                         p.text,
1212:                         p.requier,
1213:                         p.errmsg,
1214:                         p.help,
1215:                         p.regex,
1216:                     );
1217:                     vField.proctask = p.proctask;
1218:                     vField.progcall = p.progcall;
1219:                     vField.typecall = p.typecall;
1220:                     vField.parmcall = p.parmcall;
1221:                     vField.protect = p.protect;
1222:                 },
1223:
1224:                 forms.REFTYP.PASSWORD => {
1225:                     vField = fld.newFieldPassword(
1226:                         p.name,
1227:                         p.posx,
1228:                         p.posy,
1229:                         p.width,
1230:                         p.text,
1231:                         p.requier,
1232:                         p.errmsg,
1233:                         p.help,
1234:                         p.regex,
1235:                     );
1236:                     vField.proctask = p.proctask;
1237:                     vField.progcall = p.progcall;
1238:                     vField.typecall = p.typecall;
1239:                     vField.parmcall = p.parmcall;
1240:                     vField.protect = p.protect;
1241:                 },
1242:
```

```
1243:                 forms.REFTYP.YES_NO => {
1244:                     vField = fld.newFieldYesNo(
1245:                         p.name,
1246:                         p.posx,
1247:                         p.posy,
1248:                         p.text,
1249:                         p.requier,
1250:                         p.errmsg,
1251:                         p.help,
1252:                     );
1253:                     vField.proctask = p.proctask;
1254:                     vField.progcall = p.progcall;
1255:                     vField.typecall = p.typecall;
1256:                     vField.parmcall = p.parmcall;
1257:                     vField.protect = p.protect;
1258:                 },
1259:
1260:                 forms.REFTYP.SWITCH => {
1261:                     vField = fld.newFieldSwitch(
1262:                         p.name,
1263:                         p.posx,
1264:                         p.posy,
1265:                         p.zwitch,
1266:                         p.errmsg,
1267:                         p.help,
1268:                     );
1269:                     vField.proctask = p.proctask;
1270:                     vField.progcall = p.progcall;
1271:                     vField.typecall = p.typecall;
1272:                     vField.parmcall = p.parmcall;
1273:                     vField.protect = p.protect;
1274:                 },
1275:
1276:                 forms.REFTYP.DATE_FR => {
1277:                     vField = fld.newFieldDateFR(
1278:                         p.name,
1279:                         p.posx,
1280:                         p.posy,
1281:                         p.text,
1282:                         p.requier,
1283:                         p.errmsg,
1284:                         p.help,
1285:                     );
1286:                     vField.proctask = p.proctask;
1287:                     vField.progcall = p.progcall;
1288:                     vField.typecall = p.typecall;
```

```
1289:                     vField.parmcall = p.parmcall;
1290:                     vField.protect = p.protect;
1291:                 },
1292:
1293:                 forms.REFTYP.DATE_US => {
1294:                     vField = fld.newFieldDateUS(
1295:                         p.name,
1296:                         p.posx,
1297:                         p.posy,
1298:                         p.text,
1299:                         p.requier,
1300:                         p.errmsg,
1301:                         p.help,
1302:                     );
1303:                     vField.proctask = p.proctask;
1304:                     vField.progcall = p.progcall;
1305:                     vField.typecall = p.typecall;
1306:                     vField.parmcall = p.parmcall;
1307:                     vField.protect = p.protect;
1308:                 },
1309:
1310:                 forms.REFTYP.DATE_ISO => {
1311:                     vField = fld.newFieldDateISO(
1312:                         p.name,
1313:                         p.posx,
1314:                         p.posy,
1315:                         p.text,
1316:                         p.requier,
1317:                         p.errmsg,
1318:                         p.help,
1319:                     );
1320:                     vField.proctask = p.proctask;
1321:                     vField.progcall = p.progcall;
1322:                     vField.typecall = p.typecall;
1323:                     vField.parmcall = p.parmcall;
1324:                     vField.protect = p.protect;
1325:                 },
1326:
1327:                 forms.REFTYP.MAIL_ISO => {
1328:                     vField = fld.newFieldMail(
1329:                         p.name,
1330:                         p.posx,
1331:                         p.posy,
1332:                         p.width,
1333:                         p.text,
1334:                         p.requier,
```

```
1335:                         p.errmsg,
1336:                         p.help,
1337:                     );
1338:                     vField.proctask = p.proctask;
1339:                     vField.progcall = p.progcall;
1340:                     vField.typecall = p.typecall;
1341:                     vField.parmcall = p.parmcall;
1342:                     vField.protect = p.protect;
1343:                 },
1344:
1345:                 forms.REFTYP.TELEPHONE => {
1346:                     vField = fld.newFieldTelephone(
1347:                         p.name,
1348:                         p.posx,
1349:                         p.posy,
1350:                         p.width,
1351:                         p.text,
1352:                         p.requier,
1353:                         p.errmsg,
1354:                         p.help,
1355:                         p.regex,
1356:                     );
1357:                     vField.proctask = p.proctask;
1358:                     vField.progcall = p.progcall;
1359:                     vField.typecall = p.typecall;
1360:                     vField.parmcall = p.parmcall;
1361:                     vField.protect = p.protect;
1362:                 },
1363:
1364:                 forms.REFTYP.DIGIT => {
1365:                     vField = fld.newFieldDigit(
1366:                         p.name,
1367:                         p.posx,
1368:                         p.posy,
1369:                         p.width,
1370:                         p.text,
1371:                         p.requier,
1372:                         p.errmsg,
1373:                         p.help,
1374:                         p.regex,
1375:                     );
1376:                     vField.proctask = p.proctask;
1377:                     vField.progcall = p.progcall;
1378:                     vField.typecall = p.typecall;
1379:                     vField.parmcall = p.parmcall;
1380:                     vField.protect = p.protect;
```

```
1381:                 },
1382:
1383:                 forms.REFTYP.UDIGIT => {
1384:                     vField = fld.newFieldUDigit(
1385:                         p.name,
1386:                         p.posx,
1387:                         p.posy,
1388:                         p.width,
1389:                         p.text,
1390:                         p.requier,
1391:                         p.errmsg,
1392:                         p.help,
1393:                         p.regex,
1394:                     );
1395:                     vField.proctask = p.proctask;
1396:                     vField.progcall = p.progcall;
1397:                     vField.typecall = p.typecall;
1398:                     vField.parmcall = p.parmcall;
1399:                     vField.protect = p.protect;
1400:                 },
1401:
1402:                 forms.REFTYP.DECIMAL => {
1403:                     vField = fld.newFieldDecimal(
1404:                         p.name,
1405:                         p.posx,
1406:                         p.posy,
1407:                         p.width,
1408:                         p.scal,
1409:                         p.text,
1410:                         p.requier,
1411:                         p.errmsg,
1412:                         p.help,
1413:                         p.regex,
1414:                     );
1415:                     vField.proctask = p.proctask;
1416:                     vField.progcall = p.progcall;
1417:                     vField.typecall = p.typecall;
1418:                     vField.parmcall = p.parmcall;
1419:                     vField.protect = p.protect;
1420:                 },
1421:
1422:                 forms.REFTYP.UDECIMAL => {
1423:                     vField = fld.newFieldUDecimal(
1424:                         p.name,
1425:                         p.posx,
1426:                         p.posy,
```

```
1427:                         p.width,
1428:                         p.scal,
1429:                         p.text,
1430:                         p.requier,
1431:                         p.errmsg,
1432:                         p.help,
1433:                         p.regex,
1434:                     );
1435:                     vField.proctask = p.proctask;
1436:                     vField.progcall = p.progcall;
1437:                     vField.typecall = p.typecall;
1438:                     vField.parmcall = p.parmcall;
1439:                     vField.protect = p.protect;
1440:                 },
1441:
1442:                 forms.REFTYP.FUNC => {
1443:                     vField = fld.newFieldFunc(
1444:                         p.name,
1445:                         p.posx,
1446:                         p.posy,
1447:                         p.width,
1448:                         p.text,
1449:                         p.requier,
1450:                         p.procfunc,
1451:                         p.errmsg,
1452:                         p.help,
1453:                     );
1454:                     vField.proctask = p.proctask;
1455:                     vField.progcall = p.progcall;
1456:                     vField.typecall = p.typecall;
1457:                     vField.parmcall = p.parmcall;
1458:                     vField.protect = p.protect;
1459:                 },
1460:             }
1461:
1462:             vPanel.field.append(vField) catch |err| {
1463:                 @panic(@errorName(err));
1464:             };
1465:         }
1466:
1467:         for (pnlx.linev.items) |p| {
1468:             var vlinev: lnv.LINEV = undefined;
1469:
1470:             vlinev = lnv.newLine(p.name, p.posx, p.posy, p.lng, p.trace);
1471:
1472:             vPanel.linev.append(vlinev) catch |err| {
```

```
1473:                 @panic(@errorName(err));
1474:             };
1475:         }
1476:
1477:         for (pnlx.lineh.items) |p| {
1478:             var vlineh: lnh.LINEH = undefined;
1479:
1480:             vlineh = lnh.newLine(p.name, p.posx, p.posy, p.lng, p.trace);
1481:
1482:             vPanel.lineh.append(vlineh) catch |err| {
1483:                 @panic(@errorName(err));
1484:             };
1485:         }
1486:
1487:         XPANEL.append(vPanel) catch unreachable;
1488:     }
1489:
1490:     defer NPANEL.clearAndFree();
1491:
1492:
1493:     XGRID.clearRetainingCapacity();
1494:
1495:     for (NGRID.items, 0.. ) |xgrd, idx | {
1496:                 XGRID.append( grd.initGrid(
1497:                             xgrd.name,
1498:                             xgrd.posx,
1499:                             xgrd.posy,
1500:                             xgrd.pageRows,
1501:                             xgrd.separator,
1502:                             xgrd.cadre)
1503:                         ) catch |err| {@panic(@errorName(err));};
1504:
1505:              for (xgrd.cells.items ) |pcell | {
1506:                     XGRID.items[idx].cell.append(grd.CELL{
1507:                     .text = pcell.text, .reftyp = pcell.reftyp, .long = pcell.long,
1508:                     .posy = pcell.posy, .edtcar = pcell.edtcar, .atrCell = grd.toRefColor(pcell.atrCell) })
1509:                 catch |err| {@panic(@errorName(err));};
1510:             }
1511:     }
1512:
1513:
1514:
1515:     defer NGRID.clearAndFree();
1516:
1517:     XMENU.clearRetainingCapacity();
1518:
```

```
1519:     for (NMENU.items ) |xmnu| {
1520:                 XMENU.append(initMenuDef(
1521:                                     xmnu.name,
1522:                                     xmnu.posx,
1523:                                     xmnu.posy,
1524:                                     xmnu.cadre,
1525:                                     xmnu.mnuvh,
1526:                                     xmnu.xitems
1527:                                 )
1528:                             ) catch |err| {@panic(@errorName(err));};
1529:
1530:     }
1531:
1532:
1533:
1534:     defer NMENU.clearAndFree();
1535:     return;
1536: }
1537:
1538:
```