Title:	Chassis 2023 Design & Development	Created:	2023-05-21
Status:	In Development (2023-08-01)	Author:	Lion Kimbro
Related:	Technical Agenda 2023 -- M48, M100, M101		

Intent

Chassis 2023 is the successor to "Program 23," building on the lessons learned before I forfeited that effort.

It is a programming chassis written by me, Lion Kimbro, designed for the following purposes:

1. To increase efficiency in programming by providing a common backbone based on my typical assumptions, so that I don't have to program the same things over and over and over again.
2. To provide tooling around my idiosyncratic way of programming. I program exploring a radically different paradigm than I encounter in the dogmas of computer programming today. I do not believe in Object Oriented Programming. I do not believe in Functional Programming. I believe in notation, I believe in system programming, I believe in steps and promises, I believe in state, I believe in user interface, I believe in run-time shaping and data, I believe in introspection, and I believe in minimalism.
3. To make it easy to distribute programs over the Internet, composed of multiple parts, via pip. (Added: 2023-08-01)

Progress

2023-05-21	began by copying the Program 23 Design & Development document, and winnowing it down
2023-08-01	dramatic shift to package based system, rather than module based system; project now hosted in factory, rather than trigger/repo

Seed Inspirations

"TA2023" refers to "Technical Agenda 2023," my inventory and planning of project ambitions that was anticipated for a period beginning in 2023.

source	*project id*	*title*	*comments*
TA2023	M48	General Programming Framework 2023	this is more about the ideology, the philosophy that guides my programming – not about a specific tool or technology; Think: "If I were going to teach my philosophy of programming to other programmers, what would I say?" That's what I meant by "Framework." I should probably find another phrase, since that's got an overriding meaning for programmers today. Or go with "Conceptual Framework."
TA2023	M100	Standard 2023 Program Execution Framework	Previously inspired program23, now incarnated through chassis2023, this program.
TA2023	M101	Managed Module Loading	*I'm working up to this state.*
TA2023	M102	Python Call Graph Tracer	*I'm working up to this state.*

Manifest

file	*requires*	*significance*

Post-Mortem from program23

These are insights copied from the program23 documentation, intended to guide the development of chassis23:

idea	*date*	*status*	*detail*
fonts = specificity	2023-05-20		have three different fonts that grade how firm the ideas that are being written are – IBM Plex would be firm, Comic Sans would be casual, and … I'm not sure what would be in the middle
grow organically	2023-05-20		develop the system only as I make use of the system
transfer organically	2023-05-20		transfer only the capabilities that I want to make immediate use of, on an organic basis
integrate	2023-05-20		keep the concept of programdata.json, but call it package.json; but do NOT use plug-ins to the system to develop infrastructure -- instead, directly integrate the services I use (like resources, snowflake, tk23,) and turn on and off capabilities with keys and flags in package.json ; (updated 2023-08-01 for package-focality)
document less, document late	2023-05-20		minimize documentation next time around – write documentation later, when I need it, rather than up-front, or while coding
~~"chassis"~~	2023-05-20		~~the name shall be "chassis", but it'll technically be "chassis2023", and just imported as "chassis"~~ -- *2023-08-01: Now that I am working more publicly, I want to keep chassis2023, to reduce the possibility of name collision.*
~~symbols injection working early on~~	2023-05-20		~~I'll focus on its parts working, early on~~ – *2023-08-01: After some experimentation, I am abandoning symbols all-together. I found that in practice, I rarely see errors related to mistyping keys.*

in-app development	2023-05-20		I'll develop it within the program that I'm working on, early on; see if I can't use modules for the development
x5 bays	2023-05-20	COMPLETED 2023-05-21	I'll expand trigger to support x5 bays, early on, as well as bay-to-bay transfer, and direct load to bay?

Here are some problems I've noticed with this last attempt, too:

problem	*date*	*detail*
over-documenting	2023-05-20	So, I documented every function here – and that meant that, when I wanted to change the code, when I wanted to shape the code, I had to revise the documentation as well – and this started to get nightmarish… The thought of adding 5 functions, and then revising the documentation to go along with – very uncomfortable. I needed to develop code rapidly. But that seemed impossible – the documentation updates were far more intimidating than the code changes themselves. One day, I will develop a programming system that lets me visualize the code and edit the code in the same environment. But that's a ways off from the present. *Update note 2023-08-01: I recognized that documenting dictionary types helped, and documenting each function hurt, while developing. So I will try out: continuing to document dictionaries, but not documenting functions, until the program has some stability.*
parallel development pains	2023-05-20	I was trying to update program23, tk23, resources, and snowflakes, all at the same time, with a limited 3-bay system – it seems that I need more bays, or something. *Update note 2023-08-01: Integrating a lot of these systems into chassis2023 will alleviate much of the pain. And I did go to five bays.*
nested dictionary documentation	2023-05-20	It was very hard to document nested dictionaries, and I think I need a better method of documenting nested dictionaries. This may even be worthy of its own application for the detailing of.

Things that worked, too:

success	*date*	*detail*
working_office	2023-05-20	Keeping documents in a "working_office" folder, and copying them into the "working" directory before publishing a version – that worked very well for both (a) isolating Word, so that it's not holding on to the working/ directory and gumming up transfers, and (b) making it possible for me to quickly read and revise documentation between executions
documenting dictionary types	2023-05-20	This is a mixed bag, but I must say that – I feel much more confident with the dictionary types having a documented form. It was a real source of security, and worked rather well.
documenting access controls	2023-05-20	In particular, the access model here is valuable, and I think I may even start a special project just to document it.
Excalidraw	2023-05-20	Excalidraw is a fantastic system for making and embedding schematics: GUI outlines, architecture diagrams

Development Notes

2023-08-01: the Factory

After 10 years of use, I have finally graduated my previous development/repository system, "trigger.py" and "repo.py".

I am now using "factory." And "factory" is package-centric, rather than module-centric. And it also is being set up for PyPI. This adds a public scrutiny to my code, which I've been so-far shielded from. It also means that I can distribute code globally.

I had started work on "confkingdom", to make it possible to distribute parts, but quickly realized that I was essentially doing a portion of the work that is really required – which is chassis2023's work. I deprecated confkingdom, and decided to go all-in on chassis2023.

Chassis2023 will be an entire platform. Snowflake, resources, and related services will be built into it, rather than distributed piecemeal.

One thing I am struggling to figure out, is how I will do GUI configuration of packages, of programs. That's something that I really want, but I'm not sure how to make it work with Python packages. Python packages themselves seem to be a moving target. But they do seem to be able to store data.

2023-05-21: New Beginnings

I've updated trigger to support x5 bays (!), and I've fixed multiwork.

Per the new strategy, I am working on chassis2023 by working on tkinterdraw (proj:007V).

Core Themes

Concept (title)	*Date Recorded*	*Description*	*Progress*
Stages	2023-04-28	The idea is for setup, execution, and shut-down stages to be systematically pluggable. When a module is added, it should be up and running, and saving out as well, in an obvious and perhaps configurable way. If it is configurable, it is configurable through the GUI (see "GUI Configuration" concept as well.) Staging must be completely controllable by the developer: It must be possible to execute the system loading process up to a certain point, and then to stop exactly there, and then manually perform steps and analyses on the system.	Specified 2023-04-29
Promises	2023-04-28	The idea is to make it easier to explicitly note and check the status of various flags, for the sake of ensuring promise keeping. Programs can explicitly note state for ensuring promise-keeping. lock("x") can raise WasLocked, unlock("x") can raise WasNotLocked; req("x") can raise WasNotLocked and forbid("x") can raise WasLocked. There may be other forms of state notation and promise checking as well.	Specified 2023-04-28, sufficient for progress.
Call Graph Monitoring & Alerting	2023-04-28	For the sake of debugging, establishing background traces over select portions of the program shall be supported. Through tracing execution of the program, it should be easy to see which code paths are being executed, and in this way, monitor the program to see what assumptions can be made and can not be made, or are even being broken.	Needs to be looked at a little bit more.
Logging, Warning/Error Recording	2023-05-01, 2023-05-20	A system for log notes, warnings, recording errors. 2023-05-02: I may also want to identify a designated error handler, of which there is one allowed, similar to how run() works. That there are items in the errors bin is reported there, and it alone is authorized to deal with them. 2023-05-20: Extending with log notes, ring logging system, imported from pamphlet22.	2023-05-20: importing from pamphlet22.
Module Loading	2023-04-29	Automated loading of modules, modules specified in the program structure. Information about modules and configuration of modules automated.	Specified 2023-04-30, sufficient for progress.
GUI Configuration	2023-04-28	Creating a program and configuring its modules should be conducted through a GUI, and should operate relatively simply. It should be possible to do the things manually or programmatically as well, hence all configuration is stored in JSON data. The system of program execution itself reads that JSON data and uses it to load modules and start processes running. The programmer essentially programs attachments to the central system, rather than the central system itself.	Needs to be looked at, and programmed as well. NOT A BLOCKER, however.
Self-Identification	2023-04-28	The program should have a GUID, a TAGURI, a name, a description, a title, and perhaps even tags. Other modules should be able to inquire and recognize and relate the identification of the program as a whole.	Specified 2023-04-28
Typical Execution Configurations	2023-04-28	Whether the program is a CLI tool, or a Server (web, or Filetalk,) or a Text Menu based program, or a windows Tkinter program – it should be easy to setup standard base configurations, and get cranking on writing your program. 2023-04-30: This might end up just being a development tool template, or something like that. "Create a standard config for a menu-based program, ..."	Specified 2023-04-28, partly – not sure what the repercussions of the selections are, yet.
Secure Resources	2023-04-28, 2023-05-20	Resources such as JSON files, or JSON-lines files, or text files, or binary data files, and so on – the loading and saving, in an ACID transactional way, of these basic resources, should be easily configured. The use of resources.py is built-in, and use of resources can be configured in the module declaration.	Needs to be looked at.
Snowflake Services	2023-05-20	Snowflake services are built-in, and use of a snowflake system can be configured in the module declaration.	
Communications	2023-04-28	All programs should be able to easily, and simply, communicate with other such running instances. This will be accomplished through "Ding" and "Filetalk" mechanisms – "Ding" being a collection of strategies for simply alerting another process to "pay attention, something is there for you from another system," and "Filetalk" being conventions for communicating between processes through simply reading and writing JSON files kept on the filesystem.	Generative Questions about Staging

Symbols Support	2023-04-28	By default, Python does not support symbols, which are of great value for developing with categorical variables – something that I do, all of the time. When a module is loaded, the symbols that the module needs will be injected directly into the module's namespace by the Program 23 system.	Specified 2023-04-28
Symbols Documentation	2023-05-02	Allow for the definitions of symbols – and a single symbol could be defined multiple ways – and then also the grouping of symbols into collections of related symbols.	Not designed yet, though it's not hard to imagine. It'd be something like: SYMBOL: x, TITLE: y, DESC: z, and symbols would be defined in groups that have a TITLE and a DESC, and SYMBOLS: [...]. Unanswered: Where are symbols defined? In the module? Are they imported across module boundaries? This could all be specified in the program data, and with symbols imported by group name
Package Oriented	2023-08-01	The system is built around Python Packages, rather than individual Python modules.	In progress; co-developing with Factory immediately. (2023-08-01)
Factory Interlocking	2023-08-01	The system is designed to operate cooperatively with Lion's Factory system, 2023.	In progress; co-developing with Factory immediately. (2023-08-01)

Terminology

term	*date noted*	*meaning*
program23 module	2023-04-30	a module that is specifically made for use with the program23 system; includes a kMODULE_INFO block and reserves stage(), run(), and pulse() as special functions

Package File

The "package file" (formerly "program file") is the JSON file that defines a given package, authoritatively, in Lion's Factory 2023 system, and in chassis2023.

fact key	*fact value*	*information*	*date recorded*
~~program filename:~~	~~program.json~~	~~filename of the Program File~~	~~2023-04-30~~
package definition filename:	package.json	filename for the file containing definitions for the package	2023-08-01

It describes the package's self-identification, the modules it makes use of, the resources it needs in order to run, and every dimension of configuration of the package.

It does NOT describe the *user's* configuration of a program. For example, say the program is a computer game, and the user can configure the joystick and screen dimensions and other aspects like that: That's the user's configuration, and the program itself will manage the user's configuration.

Rather, this is the configuration of *the package itself*, and it is intended to be maintained and manipulated by the programmer, his or her own self.

The package file is kept in a file called `package.json`, and I absolutely do want it to be manipulated via tools. JSON does not support comments, and so "#COMMENT" keys may be used in the dictionary in order to be a place for human readable comments to be kept in the JSON dictionaries. They are to be ignored, systematically, if encountered, unless specifically displaying the comment for the developer, as a comment, is something that is valued.

I am not using TOML, because I want this file to be machine-written, and I suspect comments that are not part of the data itself, will be lost.

Date*	*Key*	*Logical Type*	*Semantic Type*	*Description*
found	APPID	k-v dictionary	APPID dictionary	
found	.GUID	str	GUID (typically v4, including dashes)	unique identifier for the package
found	.TAGURI	str	RFC 4151 TAG URI	unique identifier for the package
found	.NAME	str	lowercase [_a-z][_a-z0-9]* name	programmer-friendly identifier for the package, and the name that is used in "import" statements; contrast PACKAGE.DISTRIBUTIONNAME
found	.TITLE	str	human readable title	developer or user facing title string, for the package
found	.TAGS	str	lowercase [_a-z][_a-z0-9]* space-separated tags	list of tags describing the package; this is local to the programmer, and not intended for tagging on (say) github
found	.DESC	str	paragraph of human-readable text	a brief description of the package
found; updated 2023-08-01	PACKAGE	k-v dictionary	package definition dictionary	details of the package's definition that don't fit anywhere else; for example, the EXECUTIONTYPE (formerly titled "PROGRAM" – updated to "PACKAGE" 2023-08-01)
2023-08-03	.DISTRIBUTIONNAME	str	legal distribution package name	the name of the package, per PyPI's distribution system – it doesn't need to match the actual package name
2023-08-03	.VERSION	str	version string, by any system	the version of the package, for PyPI's distribution system
2023-08-03	.CLASSIFIERS	list of str	list of PyPI classifier strings	the classifiers that are used to locate and characterize packages in PyPI
found	.EXECUTIONTYPE	str	Categorical Values: CLITOOL, WEBSERVER, FILETALKSERVER, TKINTERGUI, INTERACTIVEMENU, LIBRARY	what kind of executable the program is
2023-08-01	.DEPENDENCIES	list of str	list of package names, locatable in PyPI	package distribution names that will be installed as dependencies, that this package relies on
2023-08-01	CHASSIS2023	k-v dictionary	chassis2023 definition dictionary	details of the program's definition that configure chassis2023's capabilities & use
found	.LOGRINGLEN	int	length of the log ring	how long the chassis2023 log ring can get
2023-08-01	FACTORY2023	k-v dictionary	factory definition dictionary	information about how the program shows up in Lion's factory system
2023-08-01	.PROJECTID	str	Factory project identity string	an ID, such as "0100", for the package, in

				Lion's factory system
2023-08-03	GITHUB	k-v dictionary or None		if the project shows up on Github, use a k-v dictionary, otherwise, just None
2023-08-03	.NAME	str	legal github project name	name of the project, on Github
found	MODULES	k-v dictionary	module inclusion dictionary	various ways of indicating the modules to include
found	RESOURCES	list of k-v dictionaries	list of resource dictionaries, per resources module	these resources will be automatically loaded and saved
2023-08-01	SNOWFLAKES	list of k-v dicts	list of snowflake definition dictionaries, per snowflakes system	these snowflake systems will be automatically loaded and saved
2023-08-01	CONFIG	list of k-v dicts	package config value dict	a list of options that can be configured for the package by the developer

(*) "found" dates – these were entered before 2023-08-01, and likely added on or around 2023-04-30.

The Module Configuration Value dictionary

Modules can specify that values are to be selected or configured by the developer, in the program configuration.

Each value to be configured is specified in this dictionary.

Key	*Logical Type*	*Semantic Type*	*Description*
NAME	str	[_a-z][_a-z0-9]* name	unique identifier (within the module) for the configuration value
TITLE	str	human readable title	developer facing title string, for the configuration value
DESC	str	paragraph of human-readable text	a brief description of the configuration value
TYPE	str	categorical values: STR, INT, FLOAT, BOOL, OPTION	a type for the value that the developer can or must enter
DEFAULT	str, int, float, bool, or None	valid value of particular type	default value when the module is added to the program, but the developer has not manually selected a value
OPTIONS	list of (str, int, float, bool, or None)	valid values of particular type	if the list is empty, it means that there are no specific options; but if the list is not empty, then the developer must select from among these options

Stages

Generative Questions about Staging.

Date of Addition	*Question*	*Approaches Inspired*	*My immediate thoughts.*
2023-04-29	How are the stages developed?	Hard-coded set, extended by manipulating the hard-code universal set.	
		Modules specify staging orders, and how they can fit together, their requirements with respect to other staging systems.	
2023-04-29	How are stages communicated?	Global variable, no variable passed to function.	Like it, but a little annoying to name a g-var in another module entirely.
		Argument to function, optionally by global variable too.	Feels redundant. And it's not quite a parameter – it really, genuinely is context. That said, it is literally different for every single call, a true parameter.
		Unique functions per stage.	Not so great: (1.) Might have to write a bunch of binding code. (2.) Takes a lot of namespace, too – each function needs to be bound.
2023-04-29	Interest registration?	Yes, via bindings.	This requires maintaining a meta block for the module.
		Yes, via function names.	This requires using a unique function per stage, in the main. It takes a bite out of your namespace.
		Yes, via function call/stage.	So, a DISCOVERY stage is called, and it requests the other events that will be handled by the stage function. An advantage of this is that as you develop the Stage function, it's easy to maintain the stages that will be answered.
		No, everything goes in one function, and it's just called unnecessarily often times.	Looks like pure downside, but one upside of it is that to find the section names, you just need to put a debug breakpoint in the one staging function, and you'll get to see all the opportunities.
2023-04-29	What are the stages of loading?	INIT, SETUP, PRELOAD, LOAD, POSTLOAD, SETUPFINAL	
		INIT, SETUP, LOAD, POSTLOAD	
		INIT, SETUP, LOAD, POSTLOAD, INTERLINK	
2023-04-29	What are the stages of execution?	PULSE	
		START, RUN	
		RUN, PULSE	Some things work on a PULSE basis, and others work on a RUN basis, and I want to keep them distinct. Still, I want to be able to have modules that have control over the pulse loop. Only one such module should be in control of it at a time. For example, tkinter will take control of the pulse loop. The menu system does not HAVE a pulse loop, at least not in the conventional sense. Perhaps the answer here is that "pulsing" *is not a stage.*
		RUN	(see note above – this is a model in which pulsing is not considered a stage – rather, it is something that happens during the RUN stage. It likely has its own function call, too

			– it is not a *stage* function call.)
		none!	Only one module may specify a run() routine. If zero or more than one modules specify a run() routine, the development configuration error is noted and presented to the user. The run routine must take care of calling program23.pulse() as well, which will call pulse() of all modules that feature pulse().
2023-04-29	What are the stages of closure?	PRECLOSE, PRESAVE, SAVE, POSTSAVE, TEARDOWN	
		PRECLOSE, SAVE, POSTSAVE, TEARDOWN	

Summary:

The specific stages are hard-coded. The current stage is stored in a global variable, program32.g[STAGE]. The stages during setup are: INIT, SETUP, LOAD, POSTLOAD, ITERLINK, and the stages during teardown are: PRECLOSE, SAVE, POSTSAVE, TEARDOWN. There is no stage for running, or for pulsing – instead, these are identified by the presence of a function in the module.

Major Stage	*Stage*	*Tasks*
Initialization	INIT	Initialize modules, populate basic tables.
	SETUP	Perform setup steps that perform actual work.
		(resources.load())
	LOAD	Load data from disk or other remote resources.
	POSTLOAD	Perform any calculations that should occur after load.
	INTERLINK	Perform any linkages with other modules' loaded data.
	DISPLAY	Perform any special display updates, assuming user interface is set up by now. For example, in a tkinter application, you might want to display a window or something here.
Tear-Down	PRECLOSE	
	SAVE	
		(resources.save())
	POSTSAVE	
	TEARDOWN	

Standard Program23 Module Functions and Variables

name	*logical and/or semantic type*	*description*	*where to learn more*
kMODULE_INFO	module info dictionary	identifies the module, declares symbols, and declares configuration options	§ "Module Info" of this document
stage()	function, no arguments, no return	reads program23.g[STAGE] and performs specific steps in preparation or shutdown	§Stages (esp: the summary) answers the main questions, enough to write and use the code
run()	function, no arguments, no return	only ONE program23 module may define this (unless it is manually imported, and not tracked by program23,)	§Stages
pulse()	function, no arguments, no return	when program23.pulse() is called, it calls every single program23 module that defines pulse	§Stages

Program23 Pseudo-Code

constant	*date added*	*value*	*what it records*	*where to learn more*
kPROGRAMDATA_FILENAME	2023-04-30	"programdata.json"	required filename for the program data required for execution	§Program File

exception	*date added*	*what it records*	*where to learn more*
BrokenPromise	2023-04-30	super-class for broken promises	§Concepts, Promises
WasLocked(BrokenPromise)	2023-04-30	something was locked, that wasn't supposed to be locked	§Concepts, Promises
WasntLocked(BrokenPromise)	2023-04-30	something wasn't locked, that was supposed to be locked	§Concepts, Promises

g-var key	*subsystem*	*date added*	*what it records*	*where to learn more*
PROGRAMDATA		2023-04-30	the programdata read out, for the current executing program;	§Program File
RUNMODULE	exec	2023-05-01	the run module, of which there must be one and only one – this is the module that defines run(); run() being similar to "main" in C	Staging, Generative Questions about Staging, "What are the Stages of Execution?"
STAGE	exec	2023-05-01	the global stage that is immediately active; read by the program23 modules to know what stage is being used; to be written to ONLY by the function perform_stage(stage)	§Stages
RINGLEN	logging	2023-05-20	the length of the logging ring	§Logging

variable	*flags*	*subsystem*	*date added*	*what it records*	*where to learn more*
locks	i	promises	2023-04-30	a set of strings, each string represents a promise	§Concepts, Promises
modules	i	exec	2023-04-30	list of program23 module's module objects that have been loaded	
pulsers	i	exec	2023-04-30	list of functions to call at pulse time	
log	PR	logging	2023-05-20	list of errors and warnings identified during execution	§Logging
ringlog	PR	logging	2023-05-20	ring list of notifications, for high-frequency notes	§Logging
noticed	PR	logging	2023-05-20	observed situations during execution	§Logging
notice_text	PRW	logging	2023-05-20	human readable descriptions of observed situations	§Logging

internal function	*flags*	*subsystem*	*date added*	*what it does*	*where to learn more*
load_programdata()	i1	program data	2023-04-30	loads from file into g[PROGRAMDATA]	
save_programdata()	i	program data	2023-04-30	saves g[PROGRAMDATA] back out to file; this is not used in typical execution – it's only used in an editor for the program data; user data should be managed by the program (or the resources system), and is NOT managed by program23	
assemble_modules_list()	i1	modules	2023-04-30	scan modules, assemble master list of modules	§Stages

register_module(module)	i	modules	2023-05-02	perform all steps to register a module – such as appending the module to the modules list, and registering pulse() fn	§Program File, Modules Inclusions
import_module_from_name(name)	I	modules	2023-05-01	add a module, it's located by name (i.e., without ".py"), rather than path	§Program File, Modules Inclusions
import_module_from_path(p)	i	modules	2023-05-01	add a module, it's located by path (i.e., a path to a file ending in ".py")	§Program File, Modules Inclusions
import_modules_from_dir(p)	i	modules	2023-05-01	add several modules, all from a directory; (*.py, *.pyc)	§Program File, Modules Inclusions
find_runmodule()	i	exec, modules	2023-05-01	find the run module, and record an error if no run module was found, or if too many were found	§Stages
register_error(code, **data)	i	errors	2023-05-01	record an error, timestamped to the present	§Logging
print_errors_report()	iu	errors	2023-05-01	crudely print out the error list	source code
log(logtype, code, title, msg)	P	logging	2023-05-20	record a log note	§Logging
ringlog(logtype, code, title, msg)	P	logging	2023-05-20	record a log note to the ring log	§Logging
print_log()	Pu	logging	2023-05-20	crudely print out the log	source code
print_ring_log()	Pu	logging	2023-05-20	crudely print out the ring log	source code
perform_prep()	ip	exec	2023-05-01	perform all preparatory work for the program23 system itself, such as loading program data and inventorying modules to be used	source code
perform_setup()	p	exec	2023-05-01	all stages for setup	§Stages
perform_run()	p	exec	2023-05-01	calls the run() function on the run module	§Stages
perform_teardown()	p	exec	2023-05-01	all stages for teardown	§Stages
perform_stage(stage)	i	exec	2023-05-01	set g[STAGE] and call all modules' stage function	§Stages

external function	*flags*	*subsystem*	*date added*	*what it does*	*where to learn more*
run()	P	exec	2023-04-30	run the program configured in the program data	
pulse()	P	exec	2023-04-30	call each function in pulsers	§Stages
lock(k)	P	promises	2023-04-30	add a key, raise trouble if it's already there	§Concepts, §§promises
unlock(k)	P	promises	2023-04-30	remove a key, raise trouble if it isn't there	"
req(k)	P	promises	2023-04-30	require that a key is there, raise trouble if it isn't	"
forbid(k)	P	promises	2023-04-30	require a key is NOT there, raise trouble if it is	"
timestamp()	Pi	util	2023-05-01	return a UTC ISO-8601 timestamp as a string	source code

Flags:

flag	*date added*	*title*	*meaning*
P	2023-04-30	public	the function or data is to be called or used by a developer, at will
p	2023-05-01	semi-public	not intended for developer use, but it *can* be used by the developer, and will be protected for that use; however, the developer *MUST* understand what they're doing, and there may be dangerous implications to developer misuse that are not adequately documented
i	2023-04-30	internal	this function is used internally to program23; on a variable, it means that it is only to be managed by internal code
1	2023-04-30	call once *only*	the system is built on the promise that this will only ever be called once
u	2023-04-30	unreliable	don't rely on this function sticking around; it'll probably go away in the future
R	2023-05-20	freely readable	this data can be freely read by anything, no efforts are made to control reading
W	2023-05-20	freely writable	this data can be freely written to by anything, no efforts are made to control what is written here

Growth Stages 2023-04-30:

1. Lump of Clay: program23
2. Create a simplistic program for basic testing.

    - menurunner.py – defines run, uses menus, does a pulse() between rounds of execution; has a configuration variable that specifies where to start
    - testmenu.py – defines a menu stage, that's run by the menurunner
3. Make it work.

Logging

Log notes are of the following types:

- NOTE – an arbitrary note, recorded mid-execution
- DBG – record for debugging purposes
- WARN – a situation that is potentially dangerous, but not immediately erroneous
- ERR – a genuine error condition

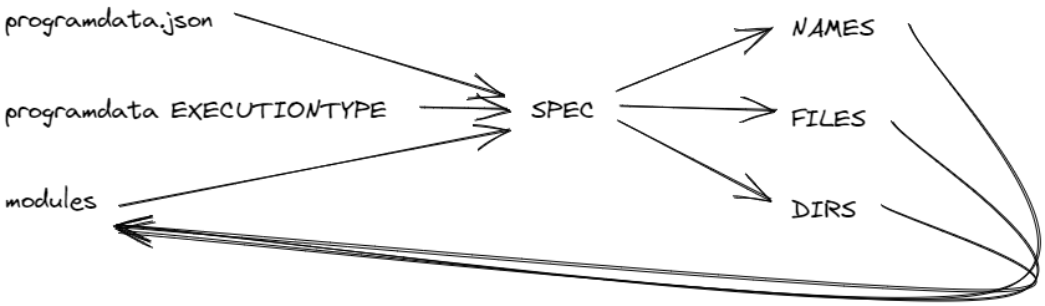
Here are some situations, and how they are classified:

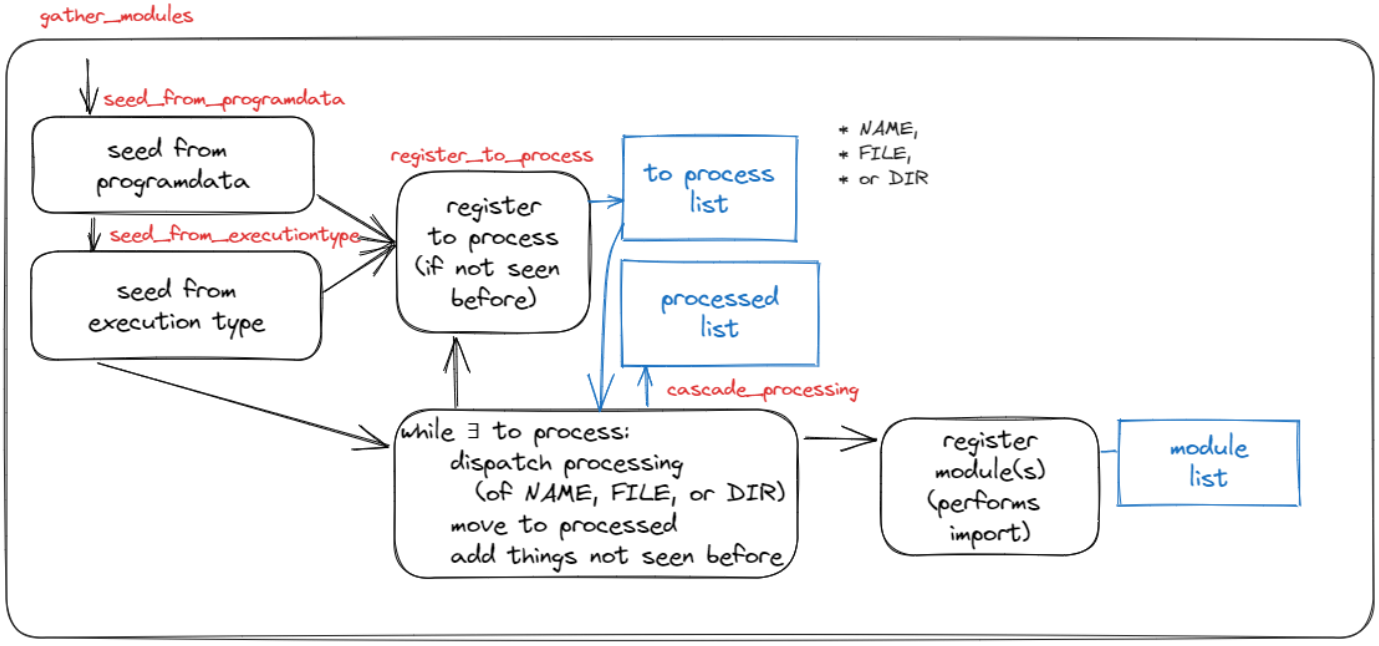
Situation	*Classification*	*Reasoning*
an error internal to program23	Raise Python Exception	program23 should be perfectly operational, and if there's a major error in there, it must be addressed
mis-configured programdata.json, blocking execution (for example, two included modules define run())	ERR	program23 should identify mis-configuration, and present the fact to the developer or user without looking like a total crash – without looking like there is something wrong within program23 itself
expected but rare circumstances that create an error situation	ERR	this isn't something wrong within program23 itself, so it should be reported to the developer or user
mis-configuration, but not necessarily blocking execution (for example, program data loaded, but no LONGRINGLEN defined)	WARN	program23 may identify mis-configuration, and present the fact to the developer or user without looking like a total crash – the situation is not immediately lethal to program execution

Log record information:

Key	*Logical Type*	*Semantic Type*	*Description*
TIME	float	ISO-8601 UTC time – sec since UNIX epoch	the time that the log record was made
SRC	list (inspect.FrameInfo)	call stack at point of recording	where the log record came from, as recorded by inspect module
TYPE	str	NOTE, DBG, WARN, or ERR	log type
CODE	str	symbol	symbol representing the specific log record type – some examples: NORUNMODULE, TOOMANYRUNMODULES
DATA	dict (sym: value)	various values that may be used elsewhere	data associated with logs of the particular coding
TITLE	str	human readable str	a title that can be presented to a human reader
DESC	str	human readable str	a description that can be presented to a human reader; Capitalize and end with a period, like a typical sentence in a book.

Module Import Process

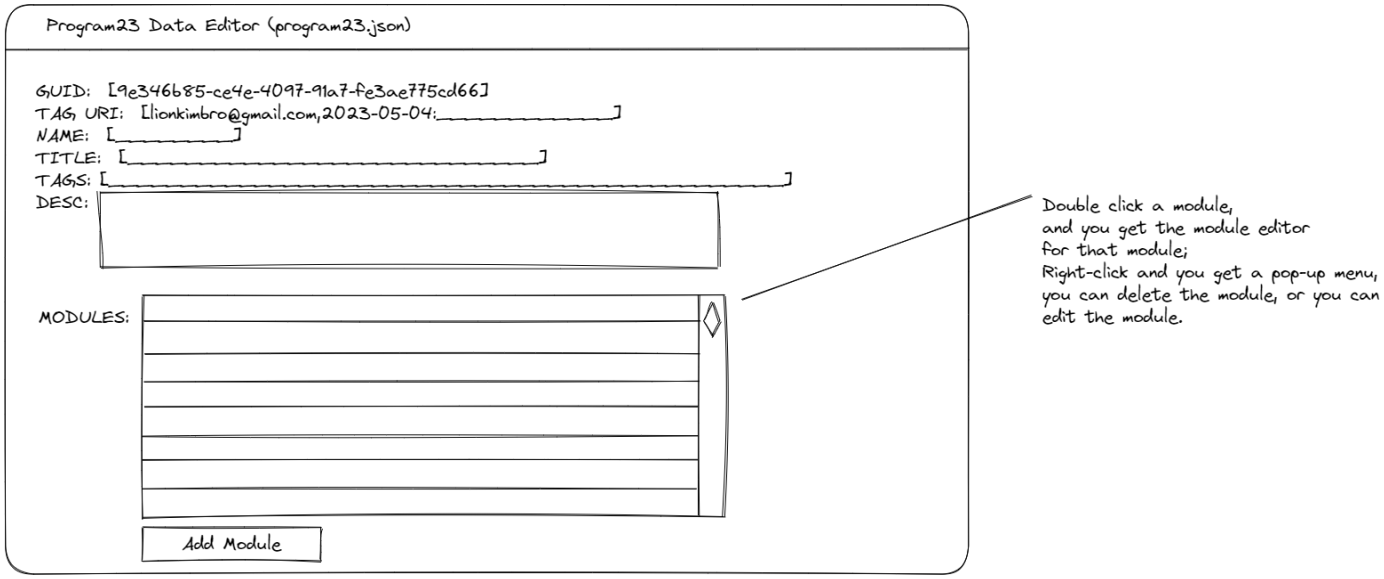




program23edit.py

What's it need to be able to do?

- Illustrate the most key data, and make it possible to edit it.
    - APPID
        - GUID – v4, autogenerated, editable
        - TAGURI – basic parts autogenerated, needs the specific name to be declared
        - NAME – prompted
        - TITLE – prompted
        - TAGS – prompted
        - DESC – prompted
    - PROGRAM
        - EXECUTIONTYPE – CLITOOL, WEBSERVER, FILETALKSERVER, TKINTERGUI, INTERACTIVEMENU
        - This might not actually work like this; this option might just go away.
- Module inclusions.
    - by NAMES, FILES, and DIRS
- Configuration of modules variables.
    - NAME (w/ TITLE) and DESCRIPTION, and a TYPE (STR, INT, FLOAT, BOOL, or OPTION), and default value, and some have OPTIONS



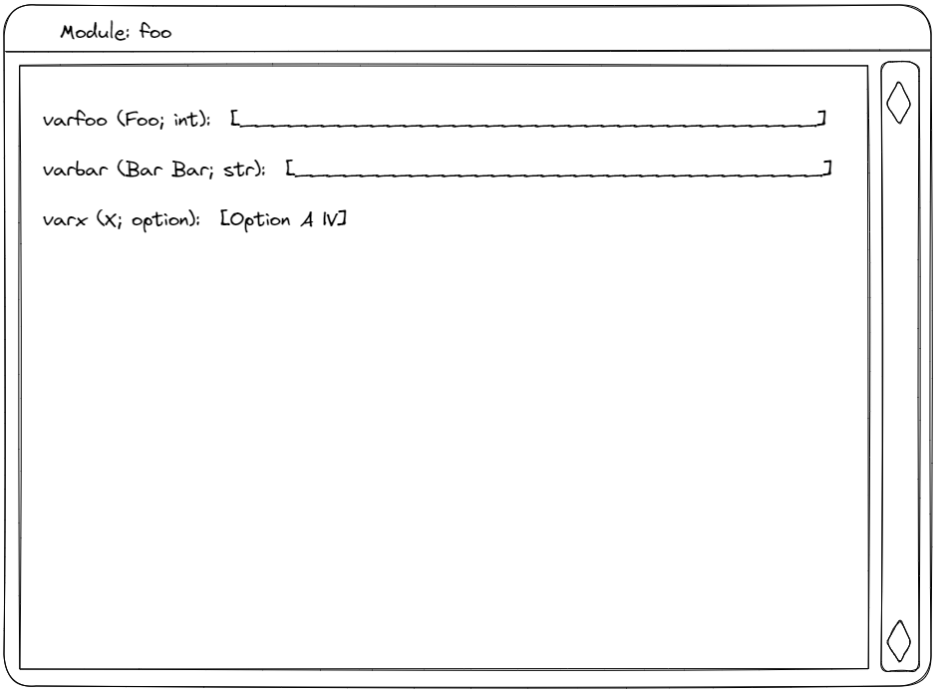
In initial development, "Add Module" will simply ask for a filename, and they will all be added by filename, relative to the current directory.

When the program closes, the data saves. The data is opened on file open, and if there is no data there, a file is created.

Simple file loading and saving services should be supported by the module structure. Program23 Data Editor, should itself work via ... the Program23 system, with the Tk system, and... the resources system? I need to remember how the Resources system works.

There's something funny about resources.py – it relies on a special resources directory. That means splitting your content between the directory that the program starts in, and a general folder that is part of a larger system. I think I want to go with the resources visision. But I need to be clear on the fact that that is what I'm doing. The vision of "where the app lives" I think needs to be adjusted for that.

I might want to look at: Where is the bin/ directory? Where are the modules kept? How do programs get installed and run? There are a lot of implications here for module packaging, delivery, installation, etc.,.


Module: foo
varfoo (Foo; int): [_____]
varbar (Bar Bar; str): [_____]
varx (X; option): [Option A IV]