# Implementation of JAT

# Introduction.

This document is used to collect various information on the implementation of JAT – the JAL to C translator. Typical topics are background information, design considerations and documentation on porting and use.

In time, much of this info will be moved to other documents.

# Status of JAT.

In the April and May of 2010, much time has been invested in the development of JAT. At the end of May, further work on JAT is postponed due to reduction of available time and in favor of research of the alternative: a C backend of the JAL compiler.

The current status of the JAT project:

* Most (or maybe all) of the JAL grammar is implementend in Antrl and translated into an Abstract Syntax Tree (AST).
* Many JAL statements are already supported (meaning: generate valid C code).
* Some constructs (like bit locations) are not supported. This means that the AST generated for this code is not checked either and could be optimized by removal of irrelevant nodes, creating extra tree levels, change grouping. Or to put it in Antlr-context: add or change rewrite rules.
* There are currently 3 embedded targets: Arduino, Arm and PIC32. The implementation proves the concept works and could be extended with exisiting C libraries, e.g. for serial comms, and an interface in JAL to use these libraries.
* For development and testing, gcc on the host system is used (Windows in my case). The test directory contains:
  + Testcases.
  + Reference output, captured from a real PIC target via the serial port, running code from the JAL compiler.
  + scripts to generate this serial output for one or all testcases.
  + Script to run the testcases on the current build of JAT and host system and compare the output with the reference.
* All control statements are implemented.
* Variable support is limited to sizes that are supported by C (byte, word, dword, sbyte, sword, sdword) .
* Simple bit variables are supported.
* The ‘mulitlpier’ for variables (bit\*2, byte\*4) are not supported yet.
* Statement in JAL are passed to C without change. This will do in most cases, but maybe not in all (especially with bit vars).
* Parameter-passing (by value, reference or JAL-volatile / ByCode) works almost all cases for the supported variables.
* Implementation of arrays has been started, but is not finished yet.

Note: implementation of arrays is mainly about passing of array parameters. This is similar to the passing of normal variables (described further in this document), but with one parameter extra.

* Variables are the most complex part of JAL and not all options are implemented yet. As mentioned before, odd-sized variables are not. Also, ‘at’ might not work (in some or all cases?).

For more details: check out the testcases – they document what works.

So where are we? Well, my IMO array support should be completed. When done, JAT is a useful tool for programming embedded systems – provided there are libraries for that target that support peripherals.

Notes:

* The statement above does not mean JAT is JAL-compatible. Especially the powerful bit-support of JAL is mostly lacking and so are odd-sized variables.
* Also, JAT is only briefly tested and probably contains quite a few bugs. Extensive (beta-)testing by multiple testers is required to achieve the production-grade level of the JAL compiler.

When continuing this development….

* Please consider to extend the testcases along with the translator. This documents all tested code and provides a way to do run a full automated regression test of all code (that should compile without errors) with one push of a button.
* You’ll see parts of the translator generate code almost one on one from the source tree. These are mainly the older parts of the code and this works well for less-complicated statements (with not too many options). Newer code tends to collect all relevant information from the AST and put this into variables. Next, the code required is generated in one step. This is more powerful and that power is required for more complex statement (e.g. a variable declaration, which has a type and multiple identifiers and each of those identifiers could contain ‘at’, but also an assignment). If ‘old-style’ code needs substantial change or extension, refactoring to the new style should be considered.

# General characteristics

* Complies as much as possible to JAL
* Is case-sensitive (unlike JAL)
* Gives transparent access to C variables
* Gives access to C functions with JAL prototypes (if function parameters are compatible)
* Supports most variables (except byte\*3) up to 32 bits

# Antlr grammar

Antlr3 C-target is used to parse the input file. To achieve this, there is a grammar file that describes the grammar of JAL at both a lexical and parser level.

At the grammar level, there are also rewrite rules, that influence the way the parsed grammar is put into the tree.

Anltr3 generates lexer and parser sourcecode that is used by JAT.

Unlike JAL, this grammar is case-sensitive (design choice – C is case-sensitive. Case sensitivity could be omitted, e.g. by converting all to lower case. This would however block access to any uppercase identifiers that exist in C.)

# Code generation

(subject to change)

Code generated in passes, walking the tree, building context.

# Parameter passing.

The JAL compiler has two ways of passing parameters.

1. In / out – The calling procedure places the calling value at \*the\* variable location of the procedure, then calls the procedure and after return, it reads the variable location and stores it in the original location. If not ‘in’, the first step is omitted (leaving the procedure with an un-initialized variable) and if not ‘out’, the last step is omitted.
2. Volatile parameters are passed like pseudo-variables: a procedure pointer is passed to read the var and one to write the var. (from/to pic\_temp? size = largest variable?)

Note: ‘Volatile’ parameters are quite different from volatile vars, supported JAL and C.

Passing method number (1) deals efficient with limitations of 16F pics and assigns specific locations to parameters and local variables. Consequence is that procedures are not re-entrant. Procedures used both in the ISR and main are duplicated and variable locations can be used for multiple procedures when they are not executed at the same time (called by and called from). Allocation of variables requires analysis for the full call tree (including pvar passing).

JAT relies on the C-mechanism of stack for parameter passing and local parameters. For ‘in’ parameters, ‘call by value’ is used and this behaves the same as code generated by the JAL compiler

For ‘in / out’ parameters (and ‘out’ parameters), ‘call by reference’ is used. As a consequence, changes are immediately applied to the original variable, while code generated by the JAL compiler only updates upon return of the procedure call and intermediate results are not propagated. This could be relevant when peripheral registers are accessed, when the original variable is accessed from ISR etc.

Volatile variables are passed as two pointers: one to a structure that contains a pointer to the put- and get-procedures and some additional parameters. The second parameter is a pointer to the variable location. This is referred to as ‘call by code’

The use of two parameters is a design choice:

* Similar variables, like bit vars can share the same structure.
* The structures are in ROM, saving valuable RAM. (This is not possible when the data-pointer was part of the structure, since both global and local variables locations are not known compile-time).

Pseudo variables are similar to volatile parameters: there is a both a structure and a pointer available. This is also the case for bit variables and other variable types that are not supported directly by C.

All call-by-code parameters are passed in 32-bits. (TODO: elaborate on why this is required)

Note: at the time of writing, only (multi-) bit variables and byte, word and dword vars are supported. Byte\*3 variables could be supported easily by means of call-by-code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **required parameter** | | |
|  |  | in | in / out | volatile |
| **Available variable** | in | Value | Create reference (pointer) at call | Pass type-struct and create reference at call |
| in / out | Dereference  at call | reference | Pass type-struct and reference at call |
| volatile | Call get-fuction at call | T.B.D.1 | code |

1 This seems to be a rare combination and a work-around at JAL source level is possible by creating a local variable, assign it, all the procedure and use the result to assign the volatile var. This feature can be implemented this in JAT as described too, but requires look-ahead (you need to create and assign a temp var before an expression and use it immediately after).