Xtc - a modern language for a classic CPU

Xtc is a programming language designed to take as much advantage of (primarily) the Atari 8-bit computer as possible. It is based on the expectation that there will be paged RAM to provide expanded memory for code or data, and it has an optimiser to squeeze as much as possible out of the 6502 processor.

Enjoy

Chapter 1. The pre-processor

The preprocessor only has a few directives, but they combine to make a useful addition to the compiler suite. The facilities of the preprocessor are more ‘meta’ than ‘language’ - they facilitate the use of the xtc language without being a part of it.

Including other files into your code

`#include <file.xt>`
will include a file, replacing the line itself with the contents of that file. The <file.xt> can also be expressed as “file.xt”, there’s no difference in operation. For example:

```
#include "data.txt"
```

`#import <file.xt>`
is the same as `#include` but guarantees that the named file will only be imported once. This means library code can be self-contained, importing what it needs, and user code can also import files without two copies ever ending up in use. For example:

```
#import <Stdio.xt>
```

Conditional compilation

`#define macro[(args)] text-string`
allows you to define macros (optionally with arguments) and use that macro later on in the source code. If a macro takes arguments, then whatever is supplied as the comma-separated arguments will have those arguments swapped in for the placeholder variables in the `#define` statement. Macros can be undefined with `#undef`. For example:

```
#define DBL(x) (double(x))
```

`#ifdef / #if / #ifndef / #elif / #else / #endif`
These allow you to control which parts of the source are compiled, or
included, or which define to use. Basically anything condition-related. For example in Math.xt, we have:

```
#ifndef ENABLE_DOUBLE
 #define ENABLE_DOUBLE 1
#endif
```

which (by default) will enable 'double' support in Maths ops, but if the user specifies -DENABLE_DOUBLE=0 on the commandline, (s)he can override that option. The ENABLE_DOUBLE is then used later in the file to decide which code is emitted

Errors and warnings

`#warning text`

allows you to insert an unconditional (unless gated by `#if` ...) warning into your code, to remind you to come back later and fix something, for example

```
#warning need to implement doFrobble()
```

`#error text`
the same as #warning but produces an error, which will stop compilation.

Variable args

The preprocessor understands macros of the form `#define myMacro(x, ...)` and will substitute-in the variable part of the macro string at instantiation if the user uses **VA_ARGS** as a string in the definition

Chapter 2. The compiler front-end

The compiler accepts several options on the command line. These can be seen by simply providing the -h or --help switch:

```
prompt% xtc -h
Usage: xtc [options] <input.xt ...>

Options:
  -a, --assemble-only        Compile to .asm module (no runtime)
  -D <name[=value]>          Define preprocessor symbol
  -E, --preprocessed <path>  Write preprocessed output to file
  -Fli, --fn-leaf-inline <n> Max leaf-function size to inline
                             (default: 100, requires -O2+)
  -Flu, --fn-loop-unroll <n> Auto-unroll counted for-loops with trip
                             count <= n (default: 5 at -O2+, 0 otherwise)
  -H, --xtc-home <path>      Set xtc home directory
  -h, --help                 Show this help
  -I <path>                  Add include search path
  -O0                        No optimisation (the default)
  -O, -O1                    Peephole + register tracking
  -O2                        + const prop, dead code/store elim, tail call
                               opt, leaf inlining, loop unrolling (<=5)
  -O3                        + branch inversion, strength reduction,
                               cross-function DCE, label cleanup
  -o, --output <path>        Output file (.asm or binary)
  -q, --quiet                Suppress informational output
  -Q, --quit-style <which>   Action after main() returns: rts (default), loop
  -S, --xtc-stack            Use xtc software stack globally

Platform options:
  -m <layout>                Load a memory layout (.lnk file). Searches
                             for <layout> as a file path (appends .lnk
                             if needed), then support/layouts/<layout>.lnk
                             Default: xl
  --dump-layout              Print memory-map diagram and exit (use with -m)
  --list-layouts, -ll        List available built-in layouts by platform

Warning options (-Wno-<category> to suppress):
    new-in-loop              new inside a loop body (leaks)
    escape                   stack-addr stored in longer-lived slot
                             (global, heap field, outer scope)
    class-init               bad initialiser on stack class
    asm-clobbers             asm{} clobbers annotation mismatch
    unknown-annotation       unrecognised function annotation
    unknown-pragma           unrecognised # directive

Function annotations (after params, comma-separated):
    : hwStack                Force 6502 hardware stack
    : naked                  No prologue/epilogue (for ISRs)
    : needsOS                Wraps body with ROM enable/disable on shadow
                             targets (no-op on non-shadow)
```

```
    : xtcStack              Force xtc software stack
                            (all annotations are case-insensitive)

Output format (determined by -o extension or [output] in .lnk):
    .asm                    6502 assembly source
    .xex .exe .bin .com     Atari XEX binary
    .prg                    C64 PRG binary

Support file search order:
    -H > $XTC_HOME > cwd > ~/xtc > /usr/local/xtc > /opt/xtc
```

Generally an invocation of xtc looks something like:

```
% xtc -Q loop -O3 ahl.xt -o test.xex
xtc: optimised -O3 (9877 → 9698 instructions)
xtc: compiled 'ahl.xt' -> 'test.xex' (0 warnings, 0 errors)
xta: assembled -> 'test.xex' (19975 bytes, 1 segments)
```

Chapter 3. Memory models

As mentioned, the compiler tries to take advantage of any banked memory available, but it supports a variety of memory layouts...

```
% xtc -ll
Available layouts (use with -m <platform>/<layout>):

  atari:
    -m atari/compy320
    -m atari/compy576
    -m atari/rambo1088
    -m atari/rambo192
    -m atari/rambo256
    -m atari/rambo320
    -m atari/rambo576
    -m atari/xe-shadow
    -m atari/xe
    -m atari/xl-shadow
    -m atari/xl
    -m atari/xt-shadow
    -m atari/xt

  commodore:
    -m commodore/c64
```

To take these in groups at a time:

- The most basic configuration of memory, not playing any tricks with shadowing ROM over RAM is 'atari/xl' (which can be shortened to just 'xl'). This represents a standard 800XL with 64KB of RAM but doesn't try to access the RAM under the OS ROM.
- The next step up is 'atari/xe' (which can be shortened to just 'xe'). This represenents a standard 130XE with 128KB of RAM. It will transparently provide access to the banked RAM, storing functions in those banks and allowing any function to call any other function naturally, without caring about the banking. Effectively you get an extra 64KB of RAM for code/data.
- The 'compyXXX' and 'ramboXXX' are effectively variants on the 'xe' which provide more memory banks. The linker scripts define which bits of PORTB to use to swap pages, in exactly the same way as the 'xe' script does. They're just "bigger xe's".
- The next step up is the -shadow variant of 'xl' or 'xe', and they disable ROM to gain access to the RAM underneath, relocate the character set so it's not in the middle of the memory block, and install a VBI interrupt handler to enable the ROM whenever there's an NMI, and disable it again when the NMI is over. If you use this, you get a much larger

code-space in "always available" RAM, but you should see the function annotation `:needsOS` discussed later.

The shadowing and banking are orthogonal and have different trade-offs ...

Shadow RAM (~14KB at $C000-$CFFF + $D800-$FFF9)

- Pros: directly addressable — no window, no paging. CPU just reads/writes. Zero overhead per access. Good for hot code, runtime routines, NMI vectors, and the unified main-code region ($A000-$CFFF, $D800-$FFF9).
- Cons: fixed size (~14KB total, split by the I/O hole). Disabling OS ROM breaks CIO, floating-point ROM, and default charset (requires the charset copy
- fake-frame VBI trampoline). Only one "bank" — you can't switch content in and out.

Banked RAM (16KB window at $4000-$7FFF, up to 1088KB total)

- Pros: huge capacity — rambo1088 gives 64+ pages. XTBankPageTracker first-fit packs classes and free functions across pages, so a large program scales well.
- Cons: every cross-bank call goes through the _xcall trampoline (save PORTB, switch, JSR, restore) — measurable per-call overhead. Only one 16KB page visible at a time; data and code on different pages can't be reached without switching. Stack, heap, and ZP pointers must stay in non-banked memory.

Rule of thumb: put hot/always-resident code (runtime, math, main, interrupt handlers) in shadow RAM; spill cold or bulky class methods into banked pages.

The compiler can be asked to produce its understanding of any memory model with the --dump-layout option, for example, the xe model with shadowing:

```
prompt% xtc --dump-layout -m xe-shadow
# xe-shadow.lnk — xe-shadow
#
# ┌──────────────────────────────────────────────────┐
# | $0082-$0083  xtc ZP: SP                          |
# ├──────────────────────────────────────────────────┤
# | $0084-$0085  xtc ZP: tmp                         |
# ├──────────────────────────────────────────────────┤
# | $0086-$0087  xtc ZP: HP                          |
# ├──────────────────────────────────────────────────┤
# | $0088-$00AF  xtc ZP: vars                        |
# ├──────────────────────────────────────────────────┤
# | $00B0-$00BF  runtime params (reserved)           |
# ├──────────────────────────────────────────────────┤
# | $00C0-$00FF  xtc ZP: vars                        |
# ├──────────────────────────────────────────────────┤
# | $2400-$3FFF  System region                       |
# |   Stack ↑ (after-system)                         |
# |   $3FFF  Heap ↓ (grows down)                     |
# ├──────────────────────────────────────────────────┤
# | $4000-$7FFF  Bank window (via $D301:$0C)         |
# ├──────────────────────────────────────────────────┤
# | $8000-$9FFF  Screen RAM                          |
# ├──────────────────────────────────────────────────┤
# | $A000-$CFFF  Main region                         |
# ├──────────────────────────────────────────────────┤
# | $D800-$FFF9  Main region                         |
# └──────────────────────────────────────────────────┘
#
# Banking: PORTB.
# Shadow mode: $D301:$01.
# Entry: $2400
```

If a layout doesn't suit your purposes, it is easy to copy/modify one of the standard ones and move code around to suit.

Chapter 4: Language specification

Types

The following types of variable are allowed in the language

Type	Meaning
u8, i8	8-bit unsigned and signed integer, respectively
u16, i16	16-bit unsigned and signed integer, respectively
u32, i32	32-bit unsigned and signed integer, respectively
float	5-byte floating point number with a 24-bit mantissa
double	8-byte floating point number with a 48-bit mantissa
bool	an alias for u8, can be set to 'true' or 'false'
string	an alias for pointer-to-u8, generally used for text
pointer	arbitrary pointer-to-type
struct	user-defined grouping of any of the above as a type
class	a struct with state and methods to call on itself

For convenience:

- We allow a series of characters in memory to be specified by enclosing within “”. Literals declared in this way are null-terminated, but the null (\0) byte is not considered to contribute to the size of the array
- Within one of those strings, we allow escape characters of
  - \n means a newline character (carriage-return + line-feed)
  - \t means a tab character
  - \r means a carriage-return character
  - \0 means an end-of-string marker
  - \\ is a literal slash () character
  - \" means a literal " character inside the quoted string
  - \' means a single ' character inside the quoted string
- We allow a single (or two, if the first character is \ ) character within ‘’ to specify the value of a single u8 byte.
- If a numeric literal has the prefix ‘$’ then it is interpreted as a hexadecimal number.
- If a numeric literal has the prefix ‘%’ then it is interpreted as a binary number.
- Any _ in a numeric literal is silently ignored

If an integer is assigned to one of smaller bit-width, then that integer will be truncated to fit into the size. No attempt will be made to sign-extend.

If a float or double is assigned to an integer, only the integral part of the float|double value will be transferred to the integer.

Casting of types is supported using the (type) construct as in ‘C’.

Types can be inferred using the ‘auto’ keyword instead of using a type explicitly. The same applies for function returns as for declarations

Syntax

The following fundamental syntax rules apply:

- Basic syntax is similar to the 'C' family of languages.
- Comments can be introduced with /* and last until */. Alternatively use // to comment up until the end of a line
- Reserved words may not be used as variables, classnames, or structure types
- Identifiers are case-sensitive, should start with a letter, and contain only letters, numbers and _
- Lines should be terminated by a semi-colon character ';'
- blocks of code can be delineated by {..} or ((..))

Variable declaration

Variables can be simple instances of the types above, structs or classes (see later), arrays, or pointers.

To declare a simple variable, the form is:

```
<type> <name> [ = <value> ];
<type> <name1, name2> [ = <value1>, <value2>];
```

eg : `u8 myVal,myOtherVal = 5,12;`

If you want to initialise the bytes that make up a value, instead of providing the natural value of a type, you can use the format:

```
<type> <name> = {byte, byte, ...};
```

and in the same manner as array initialise, any bytes that aren't provided will be inserted as 0 for you. Just as in that case, you can use [..] instead of {..}

To declare a structure, first define the structure using

```
typedef struct
    {
    u8 red;
    u8 green;
    u8 blue;
    } RGB;
```

then use it like any other variable, except that initialisation takes a list which can be either {..} or [..], for example:

```
RGB white = {255, 255, 255};
RGB black = [0,0,0];
```

An array can be declared by appending the suffix [size] to a variable's name in the declaration, for example

```
u8 bytes[32];
```

with an optional `= [value,value,…]` or `= {value, value,…}`

Pointers are used similarly to 'C' but use the @ symbol, not *, so ..

```
u8@ myPtr = &myVal;
```

... is declaring a pointer called 'myPtr' which points to the address of (the & operator returns the address) a value called 'myVal'. Pointers will be covered in more detail later.

To prevent the compiler from optimising away repeated stores to a variable, it can be declared volatile, thus in...

```
volatile u16@ dosvec = @10;
@dosvec = $1234;
@dosvec = $4567;
```

.. both assignments would happen, even at -O2 or above

To give a variable priority in zero-page allocation, declare it a 'register' variable. The compiler will scan for these throughout the source code before beginning general allocation of space. For example:

```
register u16 @myImportantPointer = $1234;
```

If a variable is declared `static` then it does not lose its availability when the scope it is defined in goes away, the previous value will be "remembered" when the function runs again. This scope can be either the file or a function-within-the-file. It is only visible within its scope

If a `static` variable is also declared `global` then it keeps its static behaviour but is visible in every scope.

Summarising variable scope and storage, we have:

Modifier	Storage	Persistence	Visibility	ZP cost
(default)	ZP	local scope	current block	yes
register	ZP (priority)	local scope	current block	yes (forced)
volatile	ZP	local scope	current block	yes
static	data section	permanent	current file / block	no
global static	data section	permanent	all files	no

It is possible to define a type to replace another with typedef, which can be useful for reducing long type definitions to shorter ones. The syntax for defining a new type is

```
typedef <type> alias;
```

Operators

In order of precedence from higher to lower, operators are very similar to the C/C++ ones, with perhaps the exception of <: and :> (which do *rotate* left and right, akin to << and >> doing *shift* left and right). The full list looks like:

Operators	Assoc	Notes
a[i], f(...), . , -> , ++ -- (postfix)	L→R	primary
+, -, !, ~, ++, -- (prefix), @ (deref), &(addr of), (type) cast, sizeof(), <, >, >>, >>> (byte extract inline asm)	R→L	unary
*, /, %	L→R	
+ -	L→R	
<:, :> (rotate)	L→R	rol, ror
<<, >> (shift)	L→R	asl, asr
<, >, <=, =>	L→R	
==, !=	L→R	

& (bitwise and)	L→R	
^ (bitwise xor)	L→R	
| (bitwise or)	L→R	
&&	L→R	
||	L→R	
? :	R→L	ternary
=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <:=, :>=	R→L	assignment

Flow control

Xct understands the following types of flow-control in programs. Note that all block-delimiters can be the traditional 'C' {..} or instead can be ((..))

- Standard C-style if/then/else. Condition is composed of the above operators in round brackets
  `if (condition) then {block} [else {block}]`
- Extended C-style switch statement. For example:

```
switch (c)
    {
    case ..12:
        // Triggers for any value <= 12
        break;

    case 13..18:
        // triggers for any value 13,14,15,16,17,18
        break;

    case 22:
        // fall through
    case 23:
        myFunction(c);
        break;

    case 40..:
        // triggers for any value >= 40
        break;

    default:
        Stdio.printf("nope\n");
        break;
    }
```

Note that we only support ranges (..x, x..y, y..) on u8 arguments - though the other decisions can be taken on any integer. Switch statements may be optimised into a jump-table at -O2 or above.

- standard 'C' "for" loop.
  `for (setup ; loop-condition ; loop-execution) {block}`
  Xct supports declaring the loop variable as part of the setup condition, so something like...
  `for (u8 i=0; i<40; i++) {..}` ... is perfectly legal
- loop over array style "for" loop
  `for ([type] var in array)` {block}
  iterates by setting var to every value in the array in turn, for example:
  `u8 chars[] = {'h', 'e', 'l', 'l', 'o'};`
  `for (u8 ch in chars) {..}`
- `while (condition) {block}`
  standard 'C' while condition, loop will execute until the condition evaluates to false

Any loop style can be exited prematurely with the `break` statement, which will jump to the end.

Equally, any loop can skip what's left of the current iteration through the loop via the `continue` statement.

If a loop-variable is declared as part of the `for(…)` statement, that variable goes out of scope once the loop terminates.

Any `for(…)` loop can be unrolled if it is lower than the unroll limit (and at least -O2 is specified), or if it is annotated with :unroll, for example:

```
for (u8 i=0; i<40; i++) :unroll
     {…}
```

Program entry

The application will start at the 'main' function, and will issue an RTS to the caller once main has finished, unless the `-Q loop` option is given to the compiler, in which case it will go into an infinite loop. The signature for main can either be

`void main(void) {…}` … or
`i16 main(u8 numArgs, string args[]) {…}`

Enumerations

the keyword `enum` introduces an enumeration, which is essentially a meaningful alias for a numeric value. The appropriately-sized type will be chosen to represent the enumeration.

Enumerations start at 0 unless otherwise specified. They increase by 1 for every element in the enumeration after the first.

Examples of enumerations might be:

```
enum suits = {hearts, clubs, diamonds, spades};
enum directions = {N = 4, S, E, W};
```

Arrays

Arrays are created with the [..] suffix on a typed name, for example:

```
u8 cakes[3];
```

Arrays can be initialised at declaration using `= [list]` or `= {list}`. If initialised, the size of the array can be inferred from the number of elements in the list, in which case no size need to be placed within the square brackets, so this is legal:

```
u8 spaces[] = {' ', '\t', '\n'};
```

Structures

Structs gather related data together in conveniently grouped packages. They are always tightly packed because the 6502 is a byte-orientated CPU. They *cannot* be named a reserved word, or have any elements that are named as reserved words.

Structs allow a dot-notation to access members within the struct, and can contain both primitive types, and other structures.

A struct can be defined as in:

```
typedef struct
   {
   u16 x;
```

```
   u8 y;
   } CursorPos;
```

Structs are value-types and have copy-semantics. They can be allocated on the stack just like any other variable in any given scope, or as globals. Structs can be passed back from functions by *value*, it is perfectly legal to do

```
  CursorPos myFunc(void)
   {
   CursorPos c = {290, 40};
   return c;
   }
```

Structs can have their address taken and passed to other functions and methods, but you cannot return a pointer to a struct outside of the scope that the struct was defined. In other words, this is ***not*** legal:

```
CursorPos@ myFunc(void)
   ((
   CursorPos c = ((290, 40));
   return &c;
   ));
```

You can, however provide a pointer to a structure as a parameter to another function in that other function's argument list, so this *is* legal:

```
CursorPos myFunc(void)
   ((
   CursorPos c = ((290, 40));
   moveCursor(&c);
   return c;
   ));
```

Structs can be assigned values to their members at initialisation, in the order in which those members are defined in the struct declaration. Any non-mentioned members will be set to 0. It is an error to provide more data-elements than are in the structure. So both of these are legal:

```
CursorPos topRight = {319};       // sets topRight to {319, 0}
CursorPos middle = {159, 100};    // sets middle to {159, 100}
```

Note that a structure can be recursively printed using the escape sequence %@ in Stdio.printf()

Pointers and address calculations

An '&' in front of an identifier's name means 'take the address of this variable', which value can be assigned to a pointer to the named identifier.

Here, an identifier can be a variable, a structure, a class, a function or another pointer, so for example, you can do:

```
pointer myVec = &myISR;
@300 = myVec;
```

Instead of the C/C++ '*', use the '@' modifier to represent "a pointer to", so ...

```
u8 @x = $580;        // x is a pointer to an 8-bit value at $580
u8 @y = &x;          // y is a pointer to the 8-bit value x
```

The construct `->` is syntactic sugar to allow easier dot-access to members within structures that you have a pointer to. This is to make things easier, because dot-syntax binds tighter than @

```
p->x === (@p).x;
```

The language allows pointer-literals, again using the @ prefix, so

```
u8 @x = @580;       // x is a pointer to a u8, and is initialised
                    // to the value in $580,$581
```

Xct also allows pointer dereferencing, so

```
u8 @x = $400;       // x is a pointer to a u8 at $400
@x = 4;             // location $400 now contains 4
```

More syntactic sugar: we allow the “type” `string` to refer to a u8@ pointer, for example:

```
string tom = “great name”;
```

For any pointer use, it does not matter if there is whitespace around the @, or whether it is placed closer to the type or the name.

Type inference

Explicit types are preferred, but the ‘auto’ keyword will infer a type if it is possible to do so. Examples..

If a function has the signature “u8 fn(void)” then the following code will infer that retVal is a u8

```
auto retVal = fn();
```

Integers can be inferred to be of the correct size based on the number of bits that they require for storage. If a number is positive it will be inferred as an unsigned int, if it is negative, it will be inferred as a signed int, so the following are valid

```
auto x = 3;         // x is a u8
auto x = -3;        // x is an i8
auto x = 257;       // x is a u16
auto x = -259;      // x is an i16
auto x = 65589;     // x is a u32
auto x = -555555;   // x is an i32
auto x = 4.5;       // x is a float
auto x = "hi";      // x is a string (u8@)
auto x = true;      // x is a bool (same for false)
```

Similarly, if a type is initialised via an expression, and that expression type can be inferred at compile time, the appropriate type can be inferred:

```
u8 x = 4;
u8 y = 5;
auto cc = x+y;      // cc = u8

u8 x = 4;
u16 y = 500;
auto cc = x+y;      // cc = u16
```

Functions

Functions are defined as in ‘C’ with a return type, although in xtc, that return type can be a tuple not just a single variable. For example:

```
(u8 x, u16 y) = myFunc();
```

```
u8,u16 myFunc(void)
    {
    return 42, 1969;
    }
```

The tuple variables can be declared at the point of calling (as above) or if existing variables are being used, the call looks like:

```
(x, y) = myFunc();
```

xtc supports recursive functions, and will store data on the xtc-stack so the 6502 stack doesn't get overwhelmed.

Functions can be exited at any point in their body using the `return` statement. If a function is defined to return a type or tuple, the `return` statement must also return a value of that type or tuple. If a function is defined to return void, then the `return` statement should have no arguments.

A function can have a variable number of arguments, declared in the usual 'C' fashion of

```
void myFunc(string fmt, …)
    {…}
```

Xtc supports C-style varargs usage, look in support/*machine*/Stdio.xt for an example of how to use these, but it follows the standard C model pretty closely.

The caller of a function can choose to inline it by prefixing the call with the inline: directive, thus removing any JSR and/or stack management. For example:

```
u8 myVal = inline:calculate(4,5);
```

Functions can be overloaded by parameter-type, so one can define several functions with the same name, which differ in what type of arguments they take, and the semantic analyser will choose the most appropriate function to actually call.

```
void show(u32 val)  { Stdio.printf("u32: %d",   val); }
void show(u8 val)   { Stdio.printf("u8 : %d",   val); }
void show(string s) { Stdio.printf("string:%s", val); }
```

Similarly, functions can be overloaded by return-type, so the semantic analyser will choose the correct function to emit from a list that might look like:

```
float myValue(void) {...}
int myValue(void) {...}
string myValue(void) {...}
```

.. depending on what variable is being assigned to the result of calling `myValue`

Functions can have annotations on them, forcing specific behaviour, for example

```
void fn(void) :naked {..}      // No register saves, just user code
void fn(void) :HwStack {..}    // Use the hardware stack
void fn(void) :xtcStack {..}   // Use the software stack
void fn(void) :needsOS {..}    // function is marked as needing OS support
void fn(void) :irq {..}        // hardware-IRQ handler, ends with RTI
void fn(void) :vbi {..}        // VBI handler, see Vbi.xt below
void fn(void) :banked {..}     // place in the bank window (xt/xe)
void fn(void) :main {..}       // place in main RAM, opt out of auto-bank
void fn(void) :shadow {..}     // place in shadow RAM (xl/xt/xe-shadow)
```

:needsOS tells the compiler that the function can't be located in shadow-RAM, and should have the ROM swapped in

before being called. Try to keep this sort of function small and self-contained. If you have many calls from a :needsOS function to functions that are stored beneath the OS, xtc will be swapping the ROM regularly, which is not efficient.

:irq marks a function as a hardware-interrupt handler. It's emitted naked (no frame save / parameter convention) and the epilogue is `RTI` instead of `RTS` so the 6502 pops the flags + return PC the IRQ pushed. Install the address into $FFFE/$FFFF (or the appropriate vector for the platform) yourself.

:vbi marks a function as a Vertical-Blank-Interrupt handler. The prologue saves A/X/Y, the body runs, the epilogue restores A/X/Y and `JMP`s through `XITVBV` ($E462) so the OS finishes the interrupt. Install via `Vbi.addImmediate(&fn)` or `Vbi.addDeferred(&fn)`; remove with `Vbi.removeImmediate()` / `Vbi.removeDeferred()`. Both routes go through `SETVBV` ($E45C) so the OS does the SEI-safe atomic write to `VVBLKI` / `VVBLKD`.

On banked targets (xt / xe) :irq and :vbi handlers are placed in main RAM at a stable address — the OS dispatcher JMPs through their vector slot directly, with no opportunity for the bank-switch trampoline to swap the right page in. The codegen handles this automatically.

:banked, :main, and :shadow control where in the address space a function lives. They're mutually exclusive. The defaults stay as they are today (auto-bank free functions on xt/xe; main RAM otherwise), so most programs don't need to think about them — they're for the cases where you want explicit control over the trade-off between fast-access main RAM, paged-and-cheaper bank window, and OS-shadow RAM.

- :banked forces the function into the bank window. On a target with no banking (xl, default Atari) it warns and falls through to :main.
- :main forces the function out of the bank window into main RAM, even on xt/xe where it would otherwise auto-bank. Useful for hot routines where the cross-bank trampoline cost matters, or for code an :irq/:vbi handler calls (since handlers can't trampoline).
- :shadow places the function under the OS ROM (the $C000-$CFFF and $D800-$FFF9 region on Atari shadow targets). On a target without shadow RAM it warns and falls through to :main. Cross-bank-style calls work either way; the only constraint is that :shadow code is unreachable when ROM is on, so don't call it from inside a :needsOS function.

Functions can be declared ahead of being defined by simply having the function signature and a ; without a body. This is the same as in C, but it's not as necessary since xct will allow functions to be called before they have been defined - it's really useful for declaring external functions in files where the function is not defined.

Functions by default will pass their parameters on the xtc software stack and use the 6502 hardware stack to handle return addresses and save registers. The `-S/--xtc-stack` commandline argument can be used to force all operations to go through the software stack (which is larger, but slightly slower).

Functions can override the commandline flags and default behaviour by being annotated with `:hwStack` or `:xtcStack`. In this case, the override wins and JSR/RTS and registers will go to whichever stack is identified. Parameters to functions will always go on the software stack, however.

Classes

Classes can define instance variables, methods, and static methods. An example of a class might be

```
class Gfx
    {
    u8 red;
    u8 green;
    u8 blue;

    void hLine(u16 x, u8 y, u8 len)
        {
        ..
        }
   }
```

Classes are often defined in their own `classname.xt` file, but this is not required.

Classes can be created in one of two ways, and it is crucial to understand the difference between them. Classes can be created on the stack or on the heap. If a class is created on the heap, *currently* it is never released - the heap only ever grows in size. If a class is created on the stack, it is reaped as soon as the scope it was created in is complete.

To create a class on the heap, use:

```
MyClass @mine = new MyClass();
```

To create a class on the stack, use

```
MyClass mine();
```

Any method in a class called 'init' that matches the parameters given by the calling code (including an init() with no parameters) will be called when the class instance is created. This uses the ability to define methods that differ only by parameter-type to offer multiple constructors of the class. There is no operator overloading, however, and no class inheritance.

If a class method is marked as `static`, then it means it can also be called without an instance. Of course, these methods can't access instance variables when called statically. A static method definition might look like:

```
class myClass
    {
    static myMethod(void)
        {..}
    }
```

Assembly Language integration

To make assembly language integration easy, the language supports an 'asm' block that can be placed anywhere in the source code. The syntax for this is:

```
asm
   {
   assembly-language instructions
   }
```

or ...

```
asm
   ((
   assembly-language instructions
   ))
```

If you create assembly as above, the compiler will determine for you which registers it thinks will be written to, and will save/restore around that. To override that determination, perhaps for wanted side-effects, you can define your own list of registers to save by telling the compiler which ones are "clobbered". The syntax for a block that clobbers all registers is:

```
asm
   {
   assembly-language instructions
   } : clobbers A,X,Y
```

Operators:

- The '<' prefix on a 16-bit or 32-bit constant or symbol evaluates to the lower byte of that constant/symbol.
- The '>' prefix on a 16-bit or 32-bit constant or symbol evaluates to bits 8..15 of that constant/symbol.

- The '>>' prefix on a 32-bit constant or symbol evaluates to bits 16..23 of that constant/symbol.
- The '>>>' prefix on a 32-bit constant or symbol evaluates to bits 24..31 of that constant/symbol.

Xtc variables

Variables will be recognised by the assembler in the same way as in the language, and can be manipulated with the operators above. For example:

```
u16 val= $1234;
asm
   {
   lda #<val;       // evaluates to LDA #$34
   ldx #>val;       // evaluates to LDX #$12
   }
```