```c
#include "crlprop.h"


/* parse programm line arguments, set-up parameter, initialise and run the simulation */
int main(int argc, const char* argv[])
{
    int ret = 0;

    double energy_keV, distance1_m, f_m, distance2_m;
    struct parameters parameters;
    struct s2c s2c;

    if (argc != 5)
    {
        fprintf(stderr, "usage: %s <energy/keV> <distance1/m> <f/m> <distance2/m>\n", argv[0]);
        ret = -1;
        goto cleanup;
    }

    energy_keV  = atof(argv[1]); /* energy in keV */
    distance1_m = atof(argv[2]); /* distance from point-source to crl device in m */
    f_m         = atof(argv[3]); /* focal distance of individual lens in m */
    distance2_m = atof(argv[4]); /* distance from crl device to detector in m */

    parameters.xray.energy        = energy_keV;
    parameters.xray.wavelength    =   12.398/energy_keV*1e-10; /* conversion factor from keV to
angstrom to metre */
    parameters.xray.wavenumber    =    2*M_PI / parameters.xray.wavelength;

    parameters.source.distance    = distance1_m;

    parameters.crl.f              = f_m;
    parameters.crl.aperture       =    1e-3; /* 1 mm */
    parameters.crl.separation     =   10e-6; /* 10 µm */
    parameters.crl.number         =    1;

    parameters.detector.distance  = distance2_m;
    parameters.detector.number    = 1000;
    parameters.detector.width     =  100e-6;
    parameters.detector.intensity = (double*) malloc(parameters.detector.number * sizeof(double));
    print_parameters(&parameters, stdout);


    /* now propagate from point-source to CRL device */
    copy_xray(  &parameters.xray,   &s2c.xray);
    copy_source(&parameters.source, &s2c.source);
    source_to_crl(&s2c);




    /* propagate through the CRL device */
    /* TODO */




    /* finally, propagte from CRL device to focus */
    /* TODO */




    /* write result to a file */
    /* TODO */




cleanup:
    return ret;
}
```

```c
/* print parameters to FILE* f; if f==NULL, print to stdout */
int print_parameters(struct parameters* para, FILE* f)
{
    if (f==NULL)
        f = stdout;

    fprintf(f, "Parameter Settings\n");
    fprintf(f, "===========================\n");
    fprintf(f, "\n");

    fprintf(f, "X-Ray Settings\n");
    fprintf(f, "---------------------------\n");
    fprintf(f, "energy:        %8.2f keV\n", para->xray.energy);
    fprintf(f, "wavelength:    %8.2f Å\n",   para->xray.wavelength*1e10);
    fprintf(f, "wavenumber:    %8.2f Å⁻¹\n", para->xray.wavenumber*1e-10);
    fprintf(f, "\n");

    fprintf(f, "Source Settings\n");
    fprintf(f, "---------------------------\n");
    fprintf(f, "distance:      %8.2f m\n",   para->source.distance);
    fprintf(f, "\n");

    fprintf(f, "CRL Device Settings\n");
    fprintf(f, "---------------------------\n");
    fprintf(f, "focal length:  %8.2f m\n",   para->crl.f);
    fprintf(f, "aperture:      %8.2f mm\n",  para->crl.aperture*1e3);
    fprintf(f, "separation:    %8.2f µm\n",  para->crl.separation*1e6);
    fprintf(f, "number/lenses: %5d\n",       para->crl.number);
    fprintf(f, "\n");

    fprintf(f, "Detector Settings\n");
    fprintf(f, "---------------------------\n");
    fprintf(f, "distance:      %8.2f m\n",   para->detector.distance);
    fprintf(f, "number/pixels: %5d\n",       para->detector.number);
    fprintf(f, "width:         %8.2f µm\n",  para->detector.width*1e6);
    fprintf(f, "pixel size:    %8.2f nm\n",  para->detector.width/para->detector.number*1e9);
    fprintf(f, "\n");

    return 0;
}


/* copy struct xray elements */
int copy_xray(struct xray* in, struct xray* out)
{
    int ret = 0;

    if (in == NULL)
    {
        fprintf(stderr, "error: in points to NULL (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }

    if (out == NULL)
    {
        fprintf(stderr, "error: out points to NULL (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }

    out->energy     = in->energy;
    out->wavelength = in->wavelength;
    out->wavenumber = in->wavenumber;

cleanup:
    return ret;
}
```

```
/* copy struct source elements */
int copy_source(struct source* in, struct source* out)
{
    int ret = 0;

    if (in == NULL)
    {
        fprintf(stderr, "error: in points to NULL (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }

    if (out == NULL)
    {
        fprintf(stderr, "error: out points to NULL (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }

    out->distance = in->distance;

cleanup:
    return ret;
}


/* propagate point-source to CRL device */
int source_to_crl(struct s2c* arg)
{
    int ret = 0;

    complex double* u = NULL;
    int N = 1000;
    int n = 1;
    int i;

    struct field field;

    if (arg == NULL)
    {
        fprintf(stderr, "error: arg points to NULL (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }

    fprintf(stderr, "warning: %s not implemented yet.\n", __FUNCTION__);

    u = (complex double*) malloc(N*sizeof(complex double));
    if (u == NULL)
    {
        fprintf(stderr, "error: malloc failed (%s:%d) [%s]\n", __FILE__, __LINE__, strerror(errno));
        ret = -1;
        goto cleanup;
    }

    /* TODO:
     * calculate proper sampling:
     * change number of points N so that phase shift between two points << 2pi

     * propagate point-source to entrance plane, save complex values in array u

     * save propagated field to file
     * calling and implementing function write_field_to_file(struct field* field);

     * save propagated field to struct field* arg->field
    */

    /* copy u to field; first get some memory */
    field.dimensions = 1;
    field.size = (int*) malloc(field.dimensions*sizeof(int));
    /* calculate total number of points in all dimensions */
    for (i=0; i<field.dimensions; i++)
    {
        field.size[i] = N;
        n *= field.size[i];
    }
    field.values = (complex double*) malloc(n*sizeof(complex double));
    /* TODO: copy u to field.values */

    /* now save the entrance field to a file */
    write_field_to_file(&field, "entrance_plane.txt");

cleanup:
    if (field.size)   free(field.size);   field.size=NULL;
    if (field.values) free(field.values); field.values=NULL;
    if (u) free(u); u=NULL;
    return ret;
}


int write_field_to_file(struct field* field, const char* fname)
{
    int ret = 0;
    FILE* f = NULL;

    if (field == NULL)
    {
        fprintf(stderr, "error: field points to NULL (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }
    if (field->dimensions < 1)
    {
        fprintf(stderr, "error: field has invalid dimensionality %d (%s:%d)\n",
                field->dimensions, __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }
    if (field->size == NULL)
    {
        fprintf(stderr, "error: field has invalid size array (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }
    if (field->values == NULL)
    {
        fprintf(stderr, "error: field has invalid values array (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }

    if (fname == NULL)
    {
        fprintf(stderr, "error: fname points to NULL (%s:%d)\n", __FILE__, __LINE__);
        ret = -1;
        goto cleanup;
    }

    f = fopen(fname, "w");
    if (f == NULL)
    {
        fprintf(stderr, "error: cannot open file [%s] (%s:%d) [%s]\n",
                fname, __FILE__, __LINE__, strerror(errno));
        ret = -1;
        goto cleanup;
    }

    fprintf(stderr, "warning: %s not implemented yet.\n", __FUNCTION__);

    /* TODO:
     * write field to FILE* f
     */

cleanup:
    if (f) fclose(f); f=NULL;
    return ret;
}
```